VISIT…

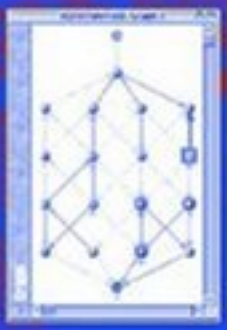
State-of-the-Art Survey

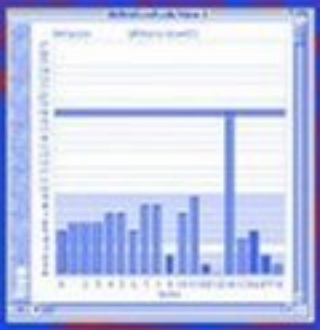
LNCS 2547

Rudolf Fleischer
Bernard Moret
Erik Meineche Schmidt (Eds.)

Experimental Algorithmics

From Algorithm Design to Robust and Efficient Software





Springer

Lecture Notes in Computer Science 2547

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Rudolf Fleischer Bernard Moret
Erik Meineche Schmidt (Eds.)

Experimental Algorithmics

From Algorithm Design to Robust and Efficient Software

Springer

Volume Editors

Rudolf Fleischer
Hong Kong University of Science and Technology
Department of Computer Science
Clear Water Bay, Kowloon, Hong Kong
E-mail: rudolf@cs.ust.hk

Bernard Moret
University of New Mexico, Department of Computer Science
Farris Engineering Bldg, Albuquerque, NM 87131-1386, USA
E-mail: moret@cs.unm.edu

Erik Meineche Schmidt
University of Aarhus, Department of Computer Science
Bld. 540, Ny Munkegade, 8000 Aarhus C, Denmark
E-mail: ems@daimi.au.dk

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>

CR Subject Classification (1998): F.2.1-2, E.1, G.1-2

ISSN 0302-9743
ISBN 3-540-00346-0 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Da-TeX Gerd Blumenstein
Printed on acid-free paper SPIN 10871673 06/3142 5 4 3 2 1 0

Preface

We are pleased to present this collection of research and survey papers on the subject of *experimental algorithmics*. In September 2000, we organized the first Schloss Dagstuhl seminar on Experimental Algorithmics (seminar no. 00371), with four featured speakers and over 40 participants. We invited some of the participants to submit write-ups of their work; these were then refereed in the usual manner and the result is now before you. We want to thank the German states of Saarland and Rhineland-Palatinate, the Dagstuhl Scientific Directorate, our distinguished speakers (Jon Bentley, David Johnson, Kurt Mehlhorn, and Bernard Moret), and all seminar participants for making this seminar a success; most of all, we thank the authors for submitting the papers that form this volume.

Experimental Algorithmics, as its name indicates, combines algorithmic work and experimentation. Thus algorithms are not just designed, but also implemented and tested on a variety of instances. In the process, much can be learned about algorithms. Perhaps the first lesson is that designing an algorithm is but the first step in the process of developing robust and efficient software for applications: in the course of implementing and testing the algorithm, many questions will invariably arise, some as challenging as those originally faced by the algorithm designer. The second lesson is that algorithm designers have an important role to play in all stages of this process, not just the original design stage: many of the questions that arise during implementation and testing are algorithmic questions—efficiency questions related to low-level algorithmic choices and cache sensitivity, accuracy questions arising from the difference between worst-case and real-world instances, as well as other, more specialized questions related to convergence rate, numerical accuracy, etc. A third lesson is the evident usefulness of implementation and testing for even the most abstractly oriented algorithm designer: implementations yield new insights into algorithmic analysis, particularly for possible extensions to current models of computation and current modes of analysis, during testing, by occasionally producing counterintuitive results, and opening the way for new conjectures and new theory.

How then do we relate "traditional" algorithm design and analysis with experimental algorithmics? Much of the seminar was devoted to this question, with presentations from nearly 30 researchers featuring work in a variety

of algorithm areas, from pure analysis to specific applications. Certain common themes emerged: practical, as opposed to theoretical, efficiency; the need to improve analytical tools so as to provide more accurate predictions of behavior in practice; the importance of *algorithm engineering*, an outgrowth of experimental algorithmics devoted to the development of efficient, portable, and reusable implementations of algorithms and data structures; and the use of experimentation in algorithm design and theoretical discovery.

Experimental algorithmics has become the focus of several workshops: *WAE*, the Workshop on Algorithm Engineering, started in 1997 and has now merged with *ESA*, the European Symposium on Algorithms, as its applied track; *ALENEX*, the Workshop on Algorithm Engineering and Experiments, started in 1998 and has since paired with *SODA*, the ACM/SIAM Symposium on Discrete Algorithms; and *WABI*, the Workshop on Algorithms in Bioinformatics, started in 2001. It is also the focus of the *ACM Journal of Experimental Algorithmics*, which published its first issue in 1996. These various forums, along with special events, such as the *DIMACS Experimental Methodology Day* in Fall 1996 (extended papers from that meeting will appear shortly in the DIMACS monograph series) and the *School on Algorithm Engineering* organized at the University of Rome in Fall 2001 (lectures by Kurt Mehlhorn, Michael Jünger, and Bernard Moret are available online at `www.info.uniroma2.it/ italiano/School/`), have helped shape the field in its formative years. A number of computer science departments now have a research laboratory in experimental algorithmics, and courses in algorithms and data structures are slowly including more experimental work in their syllabi, aided in this respect by the availability of the LEDA library of algorithms and data structures (and its associated text) and by more specialized libraries such as the CGAL library of primitives for computational geometry. Experimental algorithmics also offers the promise of more rapid and effective transfer of knowledge from academic research to industrial applications.

The articles in this volume provide a fair sampling of the work done under the broad heading of experimental algorithmics. Featured here are:

- a survey of algorithm engineering in parallel computation—an area in which even simple measurements present surprising challenges;
- an overview of visualization tools—a crucial addition to the toolkit of algorithm designers as well as a fundamental teaching tool;
- an introduction to the use of fixed-parameter formulations in the design of approximation algorithms;
- an experimental study of cache-oblivious techniques for static search trees—an awareness of the memory hierarchy has emerged over the last 10 years as a crucial element of algorithm engineering, and cache-oblivious techniques appear capable of delivering the performance of cache-aware designs without requiring a detailed knowledge of the specific architecture used;

- a novel presentation of terms, goals, and techniques for deriving asymptotic characterizations of performance from experimental data;
- a review of algorithms in VLSI designs centered on the use of binary decision diagrams (BDDs)—a concept first introduced by Claude Shannon over 50 years ago that has now become one of the main tools of VLSI design, along with a description of the BDD-Portal, a web portal designed to serve as a platform for experimentation with BDD tools;
- a quick look at two problems in computational phylogenetics—the reconstruction, from modern data, of the evolutionary tree of a group of organisms, a problem that presents special challenges in that the "correct" solution is and will forever remain unknown;
- a tutorial on how to present experimental results in a research paper;
- a discussion of several approaches to algorithm engineering for problems in distributed and mobile computing; and
- a detailed case study of algorithms for dynamic graph problems.

We hope that these articles will communicate to the reader the exciting nature of the work and help recruit new researchers to work in this emerging area.

September 2002

Rudolf Fleischer
Erik Meineche Schmidt
Bernard M.E. Moret

List of Contributors

David A. Bader
Department of Electrical and Computer Engineering
University of New Mexico
Albuquerque, NM 87131
USA
Email: dbader@eece.unm.edu
URL: http://www.eece.unm.edu/~dbader

Paul R. Cohen
Experimental Knowledge Systems Laboratory
Department of Computer Science
140 Governors Drive
University of Massachusetts, Amherst
Amherst, MA 01003-4610
USA
Email: cohen@cs.umass.edu
URL: http://www-eksl.cs.umass.edu/~cohen/home.html

Camil Demetrescu
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Italy
Email: demetres@dis.uniroma1.it
URL: http://www.dis.uniroma1.it/~demetres

Michael R. Fellows
School of Electrical Engineering and Computer Science
University of Newcastle
University Drive
Callaghan 2308
Australia
Email: mfellows@cs.newcastle.edu.au
URL: http://www.cs.newcastle.edu.au/~mfellows/index.html

Irene Finocchi
Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza"
Italy
Email: finocchi@dsi.uniroma1.it
URL: http://www.dsi.uniroma1.it/~finocchi

Rudolf Fleischer
Department of Computer Science
HKUST
Hong Kong
Email: rudolf@cs.ust.hk
URL: http://www.cs.ust.hk/~rudolf

Ray Fortna
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195
USA

Giuseppe F. Italiano
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata"
Italy
Email: italiano@info.uniroma2.it
URL: http://www.info.uniroma2.it/~italiano

Richard E. Ladner
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195
USA
Email: ladner@cs.washington.edu
URL: http://www.cs.washington.edu/homes/ladner

Catherine McGeoch
Department of Mathematics and Computer Science
Amherst College
Box 2239
Amherst, MA 01002-5000
USA
Email: ccm@cs.amherst.edu
URL: http://www.cs.amherst.edu/~ccm

Christoph Meinel
FB IV — Informatik
Universität Trier
54286 Trier
Germany
Email: meinel@uni-trier.de
URL: http://www.informatik.uni-trier.de/~meinel

Bao-Hoang Nguyen
Bernard M. E. Moret
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
USA
Email: moret@cs.unm.edu
URL: http://www.cs.unm.edu/~moret

Stefan Näher
FB IV — Informatik
Universität Trier
54286 Trier
Germany
Email: naeher@informatik.uni-trier.de
URL: http://www.informatik.uni-trier.de/~naeher

Bao-Hoang Nguyen
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195
USA

Doina Precup
Experimental Knowledge Systems Laboratory
Department of Computer Science
140 Governors Drive
University of Massachusetts, Amherst
Amherst, MA 01003-4610
USA
Email: precup-d@utcluj.ro
URL: http://www.utcluj.ro/utcn/AC/cs/pers/precup-d.html

Harald Sack
FB IV — Informatik
Universität Trier
54286 Trier
Germany
Email: sack@uni-trier.de
URL: http://www.informatik.uni-trier.de/~sack

Peter Sanders Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany
Email: sanders@mpi-sb.mpg.de
URL: http://www.mpi-sb.mpg.de/~sanders

Paul Spirakis
Computer Technology Institute, and Dept. of Computer Engineering & Informatics
University of Patras
26500 Patras
Greece
Email: spirakis@cti.gr
URL: http://kronos.cti.gr/structure/Paul_Spirakis

Christian Stangier
FB IV — Informatik
Universität Trier
54286 Trier
Germany
Email: stangier@ti.uni-trier.de
URL: http://www.informatik.uni-trier.de/~stangier

Arno Wagner
Room G64.1
Institute TIK
ETH Zürich
Gloriastr. 35
CH-8092 Zuerich
Switzerland
Email: wagner@tik.ee.ethz.ch
URL: http://www.tik.ee.ethz.ch/~wagner/

Tandy Warnow
Department of Computer Sciences
University of Texas
Austin, TX 78712
USA
Email: tandy@cs.utexas.edu
URL: http://www.cs.utexas.edu/users/tandy

Christos D. Zaroliagis
Computer Technology Institute, and Dept of Computer Engineering & Informatics
University of Patras
26500 Patras
Greece
Email: zaro@ceid.upatras.gr
URL: http://www.ceid.upatras.gr/faculty/zaro

Table of Contents

1. Algorithm Engineering for Parallel Computation
David A. Bader, Bernard M. E. Moret, and Peter Sanders 1

1.1 Introduction 1
1.2 General Issues 3
1.3 Speedup 5
1.3.1 Why Speed? 5
1.3.2 What is Speed? 5
1.3.3 Speedup Anomalies 6
1.4 Reliable Measurements 7
1.5 Test Instances 9
1.6 Presenting Results 10
1.7 Machine-Independent Measurements? 11
1.8 High-Performance Algorithm Engineering for Shared-Memory Processors 12
1.8.1 Algorithms for SMPs 12
1.8.2 Leveraging PRAM Algorithms for SMPs 13
1.9 Conclusions 15
References 15
1.A Examples of Algorithm Engineering for Parallel Computation 20

2. Visualization in Algorithm Engineering: Tools and Techniques
Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano, and Stefan Näher 24

2.1 Introduction 24
2.2 Tools for Algorithm Visualization 26
2.3 Interesting Events versus State Mapping 30
2.4 Visualization in Algorithm Engineering 33
2.4.1 Animation Systems and Heuristics: Max Flow 33
2.4.2 Animation Systems and Debugging: Spring Embedding 39
2.4.3 Animation Systems and Demos: Geometric Algorithms 41
2.4.4 Animation Systems and Fast Prototyping 43
2.5 Conclusions and Further Directions 47
References 48

3. Parameterized Complexity: The Main Ideas and Connections to Practical Computing
Michael R. Fellows 51

3.1 Introduction 51
3.2 Parameterized Complexity in a Nutshell 52
3.2.1 Empirical Motivation: Two Forms of Fixed-Parameter Complexity 52
3.2.2 The Halting Problem: A Central Reference Point 56
3.3 Connections to Practical Computing and Heuristics 58
3.4 A Critical Tool for Evaluating Approximation Algorithms . . . 64
3.5 The Extremal Connection: A General Method Relating *FPT*, Polynomial-Time Approximation, and Pre-Processing Based Heuristics 69
References 74

4. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation
Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen 78

4.1 Introduction 78
4.2 Organization 80
4.3 Cache Aware Search 80
4.4 Cache Oblivious Search 82
4.5 Program Instrumentation 84
4.6 Experimental Results 87
4.7 Conclusion 90
References 91

5. Using Finite Experiments to Study Asymptotic Performance
Catherine McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup 93

5.1 Introduction 93
5.2 Difficulties with Experimentation 97
5.3 Promising Examples 99
5.3.1 Theory with Simplifications: Writing to Parallel Disks 99
5.3.2 "Heuristic" Deduction: Random Polling 100
5.3.3 Shellsort 102
5.3.4 Sharpening a Theory: Randomized Balanced Allocation 103
5.4 Empirical Curve Bounding Rules 105
5.4.1 Guess Ratio 107
5.4.2 Guess Difference 107

5.4.3 The Power Rule . . . 108
5.4.4 The BoxCox Rule . . . 109
5.4.5 The Difference Rule . . . 110
5.4.6 Two Negative Results . . . 111
5.5 Experimental Results . . . 112
5.5.1 Parameterized Functions . . . 112
5.5.2 Algorithmic Data Sets . . . 118
5.6 A Hybrid Iterative Refinement Method . . . 120
5.6.1 Remark . . . 121
5.7 Discussion . . . 123
References . . . 124

6. WWW.BDD-Portal.ORG: An Experimentation Platform for Binary Decision Diagram Algorithms
Christoph Meinel, Harald Sack, and Arno Wagner . . . 127

6.1 Introduction . . . 127
6.1.1 WWW Portal Sites for Research Communities . . . 127
6.1.2 Binary Decision Diagrams . . . 128
6.2 A Benchmarking Platform for BDDs . . . 129
6.2.1 To Publish Code is not Optimal . . . 130
6.2.2 What is Really Needed . . . 131
6.3 A Web-Based Testbed . . . 131
6.3.1 The WWW Interface . . . 131
6.3.2 Implementation . . . 132
6.3.3 Available BDD Tools . . . 132
6.4 Added Value: A BDD Portal Site . . . 133
6.4.1 Structure of a Conventional Portal . . . 133
6.4.2 Shortcomings of Conventional Portals . . . 134
6.4.3 The BDD Portal . . . 134
6.5 Online Operation Experiences . . . 136
6.6 Related Work . . . 136
References . . . 137

7. Algorithms and Heuristics in VLSI Design
Christoph Meinel and Christian Stangier . . . 139

7.1 Introduction . . . 139
7.2 Preliminaries . . . 140
7.2.1 OBDDs – Ordered Binary Decision Diagrams . . . 140
7.2.2 Operations on OBDDs . . . 141
7.2.3 Influence of the Variable Order on the OBDD Size . . . 143
7.2.4 Reachability Analysis . . . 144
7.2.5 Image Computation Using AndExist . . . 145
7.3 Heuristics for Optimizing OBDD-Size — Variable Reordering 147
7.3.1 Sample Reordering Method . . . 147

7.3.2 Speeding up Symbolic Model Checking with Sample Sifting . . . 149
7.3.3 Experiments . . . 151
7.4 Heuristics for Optimizing OBDD Applications – Partitioned Transition Relations . . . 152
7.4.1 Common Partitioning Strategy . . . 153
7.4.2 RTL Based Partitioning Heuristic . . . 154
7.4.3 Experiments . . . 156
7.5 Conclusion . . . 157
References . . . 160

8. Reconstructing Optimal Phylogenetic Trees: A Challenge in Experimental Algorithmics
Bernard M. E. Moret and Tandy Warnow . . . 163

8.1 Introduction . . . 163
8.2 Data for Phylogeny Reconstruction . . . 165
8.2.1 Phylogenetic Reconstruction Methods . . . 166
8.3 Algorithmic and Experimental Challenges . . . 167
8.3.1 Designing for Speed . . . 167
8.3.2 Designing for Accuracy . . . 167
8.3.3 Performance Evaluation . . . 168
8.4 An Algorithm Engineering Example: Solving the Breakpoint Phylogeny . . . 168
8.4.1 Breakpoint Analysis: Details . . . 169
8.4.2 Re-Engineering BPAnalysis for Speed . . . 170
8.4.3 A Partial Assessment . . . 172
8.5 An Experimental Algorithmics Example: Quartet-Based Methods for DNA Data . . . 172
8.5.1 Quartet-Based Methods . . . 172
8.5.2 Experimental Design . . . 174
8.5.3 Some Experimental Results . . . 175
8.6 Observations and Conclusions . . . 176
References . . . 178

9. Presenting Data from Experiments in Algorithmics
Peter Sanders . . . 181

9.1 Introduction . . . 181
9.2 The Process . . . 182
9.3 Tables . . . 183
9.4 Two-Dimensional Figures . . . 184
9.4.1 The x-Axis . . . 184
9.4.2 The y-Axis . . . 187
9.4.3 Arranging Multiple Curves . . . 188
9.4.4 Arranging Instances . . . 190

9.4.5 How to Connect Measurements . . . 191
9.4.6 Measurement Errors . . . 191
9.5 Grids and Ticks . . . 192
9.6 Three-Dimensional Figures . . . 194
9.7 The Caption . . . 194
9.8 A Check List . . . 194
References . . . 195

10. Distributed Algorithm Engineering
Paul G. Spirakis and Christos D. Zaroliagis . . . 197

10.1 Introduction . . . 197
10.2 The Need of a Simulation Environment . . . 200
10.2.1 An Overview of Existing Simulation Environments . . . 202
10.3 Asynchrony in Distributed Experiments . . . 204
10.4 Difficult Input Instances for Distributed Experiments . . . 206
10.4.1 The Adversarial-Based Approach . . . 206
10.4.2 The Game-Theoretic Approach . . . 209
10.5 Mobile Computing . . . 212
10.5.1 Models of Mobile Computing . . . 213
10.5.2 Basic Protocols in the Fixed Backbone Model . . . 214
10.5.3 Basic Protocols in the Ad-Hoc Model . . . 218
10.6 Modeling Attacks in Networks:
A Useful Interplay between Theory and Practice . . . 222
10.7 Conclusion . . . 226
References . . . 226

11. Implementations and Experimental Studies of Dynamic Graph Algorithms
Christos D. Zaroliagis . . . 229

11.1 Introduction . . . 229
11.2 Dynamic Algorithms for Undirected Graphs . . . 231
11.2.1 Dynamic Connectivity . . . 231
11.2.2 Dynamic Minimum Spanning Tree . . . 243
11.3 Dynamic Algorithms for Directed Graphs . . . 252
11.3.1 Dynamic Transitive Closure . . . 252
11.3.2 Dynamic Shortest Paths . . . 264
11.4 A Software Library for Dynamic Graph Algorithms . . . 271
11.5 Conclusions . . . 273
References . . . 274

Author Index . . . 279

1. Algorithm Engineering for Parallel Computation

David A. Bader[1], Bernard M. E. Moret[1], and Peter Sanders[2]

[1] Departments of Electrical and Computer Engineering, and Computer Science
University of New Mexico, Albuquerque, NM 87131 USA
dbader@eece.unm.edu
moret@cs.unm.edu
[2] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Summary.

The emerging discipline of algorithm engineering has primarily focused on transforming pencil-and-paper *sequential* algorithms into robust, efficient, well tested, and easily used implementations. As parallel computing becomes ubiquitous, we need to extend algorithm engineering techniques to parallel computation. Such an extension adds significant complications. After a short review of algorithm engineering achievements for sequential computing, we review the various complications caused by parallel computing, present some examples of successful efforts, and give a personal view of possible future research.

1.1 Introduction

The term "algorithm engineering" was first used with specificity in 1997, with the organization of the first *Workshop on Algorithm Engineering (WAE97)*. Since then, this workshop has taken place every summer in Europe. The 1998 *Workshop on Algorithms and Experiments (ALEX98)* was held in Italy and provided a discussion forum for researchers and practitioners interested in the design, analysis and experimental testing of exact and heuristic algorithms. A sibling workshop was started in the Unites States in 1999, the *Workshop on Algorithm Engineering and Experiments (ALENEX99)*, which has taken place every winter, colocated with the *ACM/SIAM Symposium on Discrete Algorithms (SODA)*. Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and easily usable implementation. Thus it encompasses a number of topics, from modeling cache behavior to the principles of good software engineering; its main focus, however, is experimentation. In that sense, it may be viewed as a recent outgrowth of *Experimental Algorithmics* [1.54], which is specifically devoted to the development of methods, tools, and practices for assessing and refining algorithms through experimentation. The *ACM Journal of Experimental Algorithmics (JEA)*, at URL `www.jea.acm.org`, is devoted to this area.

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 1–23, 2002.
© Springer-Verlag Berlin Heidelberg 2002

High-performance algorithm engineering focuses on one of the many facets of algorithm engineering: speed. The high-performance aspect does not immediately imply parallelism; in fact, in any highly parallel task, most of the impact of high-performance algorithm engineering tends to come from refining the serial part of the code. For instance, in a recent demonstration of the power of high-performance algorithm engineering, a million-fold speedup was achieved through a combination of a 2,000-fold speedup in the serial execution of the code and a 512-fold speedup due to parallelism (a speedup, however, that will scale to any number of processors) [1.53]. (In a further demonstration of algorithm engineering, further refinements in the search and bounding strategies have added another speedup to the serial part of about 1,000, for an overall speedup in excess of 2 billion [1.55].)

All of the tools and techniques developed over the last five years for algorithm engineering are applicable to high-performance algorithm engineering. However, many of these tools need further refinement. For example, cache-efficient programming is a key to performance but it is not yet well understood, mainly because of complex machine-dependent issues like limited associativity [1.72, 1.75], virtual address translation [1.65], and increasingly deep hierarchies of high-performance machines [1.31]. A key question is whether we can find simple models as a basis for algorithm development. For example, cache-oblivious algorithms [1.31] are efficient at all levels of the memory hierarchy in theory, but so far only few work well in practice. As another example, profiling a running program offers serious challenges in a serial environment (any profiling tool affects the behavior of what is being observed), but these challenges pale in comparison with those arising in a parallel or distributed environment (for instance, measuring communication bottlenecks may require hardware assistance from the network switches or at least reprogramming them, which is sure to affect their behavior).

Ten years ago, David Bailey presented a catalog of ironic suggestions in "Twelve ways to fool the masses when giving performance results on parallel computers" [1.13], which drew from his unique experience managing the NAS Parallel Benchmarks [1.12], a set of pencil-and-paper benchmarks used to compare parallel computers on numerical kernels and applications. Bailey's "pet peeves," particularly concerning abuses in the reporting of performance results, are quite insightful. (While some items are technologically outdated, they still prove useful for comparisons and reports on parallel performance.) We rephrase several of his observations into guidelines in the framework of the broader issues discussed here, such as accurately measuring and reporting the details of the performed experiments, providing fair and portable comparisons, and presenting the empirical results in a meaningful fashion.

This paper is organized as follows. Section 1.2 introduces the important issues in high-performance algorithm engineering. Section 1.3 defines terms and concepts often used to describe and characterize the performance of parallel algorithms in the literature and discusses anomalies related to parallel

speedup. Section 1.4 addresses the problems involved in fairly and reliably measuring the execution time of a parallel program—a difficult task because the processors operate asynchronously and thus communicate nondeterministically (whether through shared-memory or interconnection networks), Section 1.5 presents our thoughts on the choice of test instances: size, class, and data layout in memory. Section 1.6 briefly reviews the presentation of results from experiments in parallel computation. Section 1.7 looks at the possibility of taking truly machine-independent measurements. Finally, Section 1.8 discusses ongoing work in high-performance algorithm engineering for symmetric multiprocessors that promises to bridge the gap between the theory and practice of parallel computing. In an appendix, we briefly discuss ten specific examples of published work in algorithm engineering for parallel computation.

1.2 General Issues

Parallel computer architectures come in a wide range of designs. While any given parallel machine can be classified in a broad taxonomy (for instance, as distributed memory or shared memory), experience has shown that each platform is unique, with its own artifacts, constraints, and enhancements. For example, the Thinking Machines CM-5, a distributed-memory computer, is interconnected by a fat-tree data network [1.48], but includes a separate network that can be used for fast barrier synchronization. The SGI Origin [1.47] provides a global address space to its shared memory; however, its nonuniform memory access requires the programmer to handle data placement for efficient performance. Distributed-memory cluster computers today range from low-end Beowulf-class machines that interconnect PC computers using commodity technologies like Ethernet [1.18, 1.76] to high-end clusters like the NSF Terascale Computing System at Pittsburgh Supercomputing Center, a system with 750 4-way AlphaServer nodes interconnected by Quadrics switches.

Most modern parallel computers are programmed in single-program, multiple-data (SPMD) style, meaning that the programmer writes one program that runs concurrently on each processor. The execution is specialized for each processor by using its processor identity (id or rank). Timing a parallel application requires capturing the elapsed wall-clock time of a program (instead of measuring CPU time as is the common practice in performance studies for sequential algorithms). Since each processor typically has its own clock, timing suite, or hardware performance counters, each processor can only measure its own view of the elapsed time or performance by starting and stopping its own timers and counters.

High-throughput computing is an alternative use of parallel computers whose objective is to maximize the number of independent jobs processed per

unit of time. Condor [1.49], Portable Batch System (PBS) [1.56], and Load-Sharing Facility (LSF) [1.62] are examples of available queuing and scheduling packages that allow a user to easily broker tasks to compute farms and to various extents balance the resource loads, handle heterogeneous systems, restart failed jobs, and provide authentication and security. *High-performance* computing, on the other hand, is primarily concerned with optimizing the speed at which a single task executes on a parallel computer. For the remainder of this paper, we focus entirely on high-performance computing that requires non-trivial communication among the running processors.

Interprocessor communication often contributes significantly to the total running time. In a cluster, communication typically uses data networks that may suffer from congestion, nondeterministic behavior, routing artifacts, etc. In a shared-memory machine, communication through coordinated reads from and writes to shared memory can also suffer from congestion, as well as from memory coherency overheads, caching effects, and memory subsystem policies. Guaranteeing that the repeated execution of a parallel (or even sequential!) program will be identical to the prior execution is impossible in modern machines, because the state of each cache cannot be determined *a priori*—thus affecting relative memory access times—and because of nondeterministic ordering of instructions due to out-of-order execution and runtime processor optimizations.

Parallel programs rely on communication layers and library implementations that often figure prominently in execution time. Interprocessor messaging in scientific and technical computing predominantly uses the Message-Passing Interface (MPI) standard [1.51], but the performance on a particular platform may depend more on the implementation than on the use of such a library. MPI has several implementations as open source and portable versions such as MPICH [1.33] and LAM [1.60], as well as native, vendor implementations from Sun Microsystems and IBM. Shared-memory programming may use POSIX threads [1.64] from a freely-available implementation (e.g., [1.57]) or from a commercial vendor's platform. Much attention has been devoted lately to OpenMP [1.61], a standard for compiler directives and runtime support to reveal algorithmic concurrency and thus take advantage of shared-memory architectures; once again, implementations of OpenMP are available both in open source and from commercial vendors. There are also several higher-level parallel programming abstractions that use MPI, OpenMP, or POSIX threads, such as implementations of the Bulk-Synchronous Parallel (BSP) model [1.77, 1.43, 1.22] and data-parallel languages like High-Performance Fortran [1.42]. Higher-level application framework such as KeLP [1.29] and POOMA [1.27] also abstract away the details of the parallel communication layers. These frameworks enhance the expressiveness of data-parallel languages by providing the user with a high-level programming abstraction for block-structured scientific calculations. Using object-oriented techniques, KeLP and POOMA contain runtime support for

non-uniform domain decomposition that takes into consideration the two main levels (intra- and inter-node) of the memory hierarchy.

1.3 Speedup

1.3.1 Why Speed?

Parallel computing has two closely related main uses. First, with more memory and storage resources than available on a single workstation, a parallel computer can solve correspondingly larger instances of the same problems. This increase in size can translate into running higher-fidelity simulations, handling higher volumes of information in data-intensive applications (such as long-term global climate change using satellite image processing [1.83]), and answering larger numbers of queries and datamining requests in corporate databases. Secondly, with more processors and larger aggregate memory subsystems than available on a single workstation, a parallel computer can often solve problems faster. This increase in speed can also translate into all of the advantages listed above, but perhaps its crucial advantage is in turnaround time. When the computation is part of a real-time system, such as weather forecasting, financial investment decision-making, or tracking and guidance systems, turnaround time is obviously the critical issue. A less obvious benefit of shortened turnaround time is higher-quality work: when a computational experiment takes less than an hour, the researcher can afford the luxury of exploration—running several different scenarios in order to gain a better understanding of the phenomena being studied.

1.3.2 What is Speed?

With sequential codes, the performance indicator is running time, measured by CPU time as a function of input size. With parallel computing we focus not just on running time, but also on how the additional resources (typically processors) affect this running time. Questions such as "does using twice as many processors cut the running time in half?" or "what is the maximum number of processors that this computation can use efficiently?" can be answered by plots of the performance *speedup*. The *absolute speedup* is the ratio of the running time of the fastest known sequential implementation to that of the parallel running time. The fastest parallel algorithm often bears little resemblance to the fastest sequential algorithm and is typically much more complex; thus running the parallel implementation on one processor often takes much longer than running the sequential algorithm—hence the need to compare to the sequential, rather than the parallel, version. Sometimes, the parallel algorithm reverts to a good sequential algorithm if the number of processors is set to one. In this case it is acceptable to report *relative speedup*, i.e., the speedup of the p-processor version relative to the 1-processor

version of the same implementation. But even in that case, the 1-processor version must make all of the obvious optimizations, such as eliminating unnecessary data copies between steps, removing self communications, skipping precomputing phases, removing collective communication broadcasts and result collection, and removing all locks and synchronizations. Otherwise, the relative speedup may present an exaggeratedly rosy picture of the situation. *Efficiency*, the ratio of the speedup to the number of processors, measures the effective use of processors in the parallel algorithm and is useful when determining how well an application scales on large numbers of processors. In any study that presents speedup values, the methodology should be clearly and unambiguously explained—which brings us to several common errors in the measurement of speedup.

1.3.3 Speedup Anomalies

Occasionally so-called *superlinear* speedups, that is, speedups greater than the number of processors,[1] cause confusion because such should not be possible by Brent's principle (a single processor can simulate a p-processor algorithm with a uniform slowdown factor of p). Fortunately, the sources of "superlinear" speedup are easy to understand and classify.

Genuine superlinear absolute speedup can be observed without violating Brent's principle if the space required to run the code on the instance exceeds the memory of the single-processor machine, but not that of the parallel machine. In such a case, the sequential code swaps to disk while the parallel code does not, yielding an enormous and entirely artificial slowdown of the sequential code. On a more modest scale, the same problem could occur one level higher in the memory hierarchy, with the sequential code constantly cache-faulting while the parallel code can keep all of the required data in its cache subsystems.

A second reason is that the running time of the algorithm strongly depends on the particular input instance and the number of processors. For example, consider searching for a given element in an unordered array of $n \gg p$ elements. The sequential algorithm simply examines each element of the array in turn until the given element is found. The parallel approach may assume that the array is already partitioned evenly among the processors and has each processor proceed as in the sequential version, but using only its portion of the array, with the first processor to find the element halting the execution. In an experiment in which the item of interest always lies in position $n - n/p + 1$, the sequential algorithm always takes $n - n/p$ steps, while the parallel algorithm takes only *one* step, yielding a relative speedup of $n - n/p \gg p$. Although strange, this speedup does not violate Brent's principle, which only makes claims on the absolute speedup. Furthermore, such strange effects often disappear if one averages over all inputs. In the example

[1] Strictly speaking, "efficiency larger than one" would be the better term.

of array search, the sequential algorithm will take an expected $n/2$ steps and the parallel algorithm $n/(2p)$ steps, resulting in a speedup of p on average.

However, this strange type of speedup does *not* always disappear when looking at all inputs. A striking example is random search for satisfying assignments of a propositional logical formula in 3-CNF (conjunctive normal form with three literals per clause): Start with a random assignment of truth values to variables. In each step pick a random violated clause and make it satisfied by flipping a bit of a random variable appearing in it. Concerning the best upper bounds for its sequential execution time, little good can be said. However, Schöning [1.74] shows that one gets exponentially better expected execution time bounds if the algorithm is run in parallel for a huge number of (simulated) processors. In fact, the algorithm remains the fastest known algorithm for 3-SAT, exponentially faster than any other known algorithm. Brent's principle is not violated since the best sequential algorithm turns out to be the emulation of the parallel algorithm. The lesson one can learn is that parallel algorithms might be a source of good sequential algorithms too.

Finally, there are many cases were superlinear speedup is not genuine. For example, the sequential and the parallel algorithms may not be applicable to the same range of instances, with the sequential algorithm being the more general one—it may fail to take advantage of certain properties that could dramatically reduce the running time or it may run a lot of unnecessary checking that causes significant overhead. For example, consider sorting an unordered array. A sequential implementation that works on every possible input instance cannot be fairly compared with a parallel implementation that makes certain restrictive assumptions—such as assuming that input elements are drawn from a restricted range of values or from a given probability distribution, etc.

1.4 Reliable Measurements

The performance of a parallel algorithm is characterized by its running time as a function of the input data and machine size, as well as by derived measures such as speedup. However, measuring running time in a fair way is considerably more difficult to achieve in parallel computation than in serial computation.

In experiments with serial algorithms, the main variable is the choice of input datasets; with parallel algorithms, another variable is the machine size. On a single processor, capturing the execution time is simple and can be done by measuring the time spent by the processor in executing instructions from the user code—that is, by measuring *CPU time*. Since computation includes memory access times, this measure captures the notion of "efficiency" of a serial program—and is a much better measure than *elapsed wall-clock time* (using a system clock like a stopwatch), since the latter is affected by all other processes running on the system (user programs, but also system routines,

interrupt handlers, daemons, etc.) While various structural measures help in assessing the behavior of an implementation, the CPU time is the definitive measure in a serial context [1.54].

In parallel computing, on the other hand, we want to measure how long the entire parallel computer is kept busy with a task. A parallel execution is characterized by the time elapsed from the time the first processor started working to the time the last processor completed, so we cannot measure the time spent by just one of the processors—such a measure would be unjustifiably optimistic! In any case, because data communication between processors is not captured by CPU time and yet is often a significant component of the parallel running time, we need to measure not just the time spent executing user instructions, but also waiting for barrier synchronizations, completing message transfers, and any time spent in the operating system for message handling and other ancillary support tasks. For these reasons, the use of elapsed wall-clock time is mandatory when testing a parallel implementation. One way to measure this time is to synchronize all processors after the program has been started. Then one processor starts a timer. When the processors have finished, they synchronize again and the processor with the timer reads its content.

Of course, because we are using elapsed wall-clock time, other running programs on the parallel machine will inflate our timing measurements. Hence, the experiments must be performed on an otherwise unloaded machine, by using dedicated job scheduling (a standard feature on parallel machines in any case) and by turning off unnecessary daemons on the processing nodes. Often, a parallel system has "lazy loading" of operating system facilities or one-time initializations the first time a specific function is called; in order not to add the cost of these operations to the running time of the program, several warm-up runs of the program should be made (usually internally within the executable rather than from an external script) before making the timing runs.

In spite of these precautions, the average running time might remain irreproducible. The problem is that, with a large number of processors, one processor is often delayed by some operating system event and, in a typical tightly synchronized parallel algorithm, the entire system will have to wait. Thus, even rare events can dominate the execution time, since their frequency is multiplied by the number of processors. Such problems can sometimes be uncovered by producing many fine-grained timings in many repetitions of the program run and then inspecting the histogram of execution times. A standard technique to get more robust estimates for running times than the average is to take the median. If the algorithm is randomized, one must first make sure that the execution time deviations one is suppressing are really caused by external reasons. Furthermore, if individual running times are not at least two to three orders of magnitude larger than the clock resolution,

one should not use the median but the average of a filtered set of execution times where the largest and smallest measurements have been thrown out.

When reporting running times on parallel computers, all relevant information on the platform, compilation, input generation, and testing methodology, must be provided to ensure repeatability (in a statistical sense) of experiments and accuracy of results.

1.5 Test Instances

The most fundamental characteristic of a scientific experiment is reproducibility. Thus the instances used in a study must be made available to the community. For this reason, a common format is crucial. Formats have been more or less standardized in many areas of Operations Research and Numerical Computing. The DIMACS Challenges have resulted in standardized formats for many types of graphs and networks, while the library of Traveling Salesperson instances, TSPLIB, has also resulted in the spread of a common format for TSP instances. The CATS project [1.32] aims at establishing a collection of benchmark datasets for combinatorial problems and, incidentally, standard formats for such problems.

A good collection of datasets must consist of a mix of real and generated (artificial) instances. The former are of course the "gold standard," but the latter help the algorithm engineer in assessing the weak points of the implementation with a view to improving it. In order to provide a real test of the implementation, it is essential that the test suite include sufficiently large instances. This is particularly important in parallel computing, since parallel machines often have very large memories and are almost always aimed at the solution of large problems; indeed, so as to demonstrate the efficiency of the implementation for a large number of processors, one sometimes has to use instances of a size that exceeds the memory size of a uniprocessor. On the other hand, abstract asymptotic demonstrations are not useful: there is no reason to run artificially large instances that clearly exceed what might arise in practice over the next several years. (Asymptotic analysis can give us fairly accurate predictions for very large instances.) Hybrid problems, derived from real datasets through carefully designed random permutations, can make up for the dearth of real instances (a common drawback in many areas, where commercial companies will not divulge the data they have painstakingly gathered).

Scaling the datasets is more complex in parallel computing than in serial computing, since the running time also depends on the number of processors. A common approach is to scale up instances linearly with the number of processors; a more elegant and instructive approach is to scale the instances so as to keep the efficiency constant, with a view to obtain isoefficiency curves.

A vexing question in experimental algorithmics is the use of worst-case instances. While the design of such instances may attract the theoretician

(many are highly nontrivial and often elegant constructs), their usefulness in characterizing the practical behavior of an implementation is dubious. Nevertheless, they do have a place in the arsenal of test sets, as they can test the robustness of the implementation or the entire system—for instance, an MPI implementation can succumb to network congestion if the number of messages grows too rapidly, a behavior that can often be triggered by a suitably crafted instance.

1.6 Presenting Results

Presenting experimental results for high-performance algorithm engineering should follow the principles used in presenting results for sequential computing. But there are additional difficulties. One gets an additional parameter with the number of processors used and parallel execution times are more platform dependent. McGeoch and Moret discuss the presentation of experimental results in the article "How to Present a Paper on Experimental Work with Algorithms" [1.50]. The key entries include

- describe and motivate the specifics of the experiments
- mention enough details of the experiments (but do not mention too many details)
- draw conclusions and support them (but make sure that the support is real)
- use graphs, not tables—a graph is worth a thousand table entries
- use suitably normalized scatter plots to show trends (and how well those trends are followed)
- explain what the reader is supposed to see

This advice applies unchanged to the presentation of high-performance experimental results. A summary of more detailed rules for preparing graphs and tables can also be found in this volume.

Since the main question in parallel computing is one of scaling (with the size of the problem or with the size of the machine), a good presentation needs to use suitable preprocessing of the data to demonstrate the key characteristics of scaling in the problem at hand. Thus, while it is always advisable to give some absolute running times, the more useful measure will be speedup and, better, efficiency. As discussed under testing, providing an *ad hoc* scaling of the instance size may reveal new properties: scaling the instance with the number of processors is a simple approach, while scaling the instance to maintain constant efficiency (which is best done after the fact through sampling of the data space) is a more subtle approach.

If the application scales very well, efficiency is clearly preferable to speedup, as it will magnify any deviation from the ideal linear speedup: one can use a logarithmic scale on the horizontal scale without affecting the legibility of the graph—the ideal curve remains a horizontal at ordinate 1.0,

whereas log-log plots tend to make everything appear linear and thus will obscure any deviation. Similarly, an application that scales well will give very monotonous results for very large input instances—the asymptotic behavior was reached early and there is no need to demonstrate it over most of the graph; what does remain of interest is how well the application scales with larger numbers of processors, hence the interest in efficiency. The focus should be on characterizing efficiency and pinpointing any remaining areas of possible improvement.

If the application scales only fairly, a scatter plot of speedup values as a function of the sequential execution time can be very revealing, as poor speedup is often data-dependent. Reaching asymptotic behavior may be difficult in such a case, so this is the right time to run larger and larger instances; in contrast, isoefficiency curves are not very useful, as very little data is available to define curves at high efficiency levels. The focus should be on understanding the reasons why certain datasets yield poor speedup and others good speedup, with the goal of designing a better algorithm or implementation based on these findings.

1.7 Machine-Independent Measurements?

In algorithm engineering, the aim is to present repeatable results through experiments that apply to a broader class of computers than the specific make of computer system used during the experiment. For sequential computing, empirical results are often fairly machine-independent. While machine characteristics such as word size, cache and main memory sizes, and processor and bus speeds differ, comparisons across different uniprocessor machines show the same trends. In particular, the number of memory accesses and processor operations remains fairly constant (or within a small constant factor).

In high-performance algorithm engineering with parallel computers, on the other hand, this portability is usually absent: each machine and environment is its own special case. One obvious reason is major differences in hardware that affect the balance of communication and computation costs—a true shared-memory machine exhibits very different behavior from that of a cluster based on commodity networks.

Another reason is that the communication libraries and parallel programming environments (e.g., MPI [1.51], OpenMP [1.61], and High-Performance Fortran [1.42]), as well as the parallel algorithm packages (e.g., fast Fourier transforms using FFTW [1.30] or parallelized linear algebra routines in ScaLAPACK [1.24]), often exhibit differing performance on different types of parallel platforms. When multiple library packages exist for the same task, a user may observe different running times for each library version even on the same platform. Thus a running-time analysis should clearly separate the time spent in the user code from that spent in various library calls. Indeed, if particular library calls contribute significantly to the running time, the

number of such calls and running time for each call should be recorded and used in the analysis, thereby helping library developers focus on the most cost-effective improvements. For example, in a simple message-passing program, one can characterize the work done by keeping track of sequential work, communication volume, and number of communications. A more general program using the collective communication routines of MPI could also count the number of calls to these routines. Several packages are available to instrument MPI codes in order to capture such data (e.g., MPICH's nupshot [1.33], Pablo [1.66], and Vampir [1.58]). The SKaMPI benchmark [1.69] allows running-time predictions based on such measurements even if the target machine is not available for program development. For example, one can check the page of results[2] or ask a customer to run the benchmark on the target platform. SKaMPI was designed for robustness, accuracy, portability, and efficiency. For example, SKaMPI adaptively controls how often measurements are repeated, adaptively refines message-length and step-width at "interesting" points, recovers from crashes, and automatically generates reports.

1.8 High-Performance Algorithm Engineering for Shared-Memory Processors

Symmetric multiprocessor (SMP) architectures, in which several (typically 2 to 8) processors operate in a true (hardware-based) shared-memory environment and are packaged as a single machine, are becoming commonplace. Most high-end workstations are available with dual processors and some with four processors, while many of the new high-performance computers are clusters of SMP nodes, with from 2 to 64 processors per node. The ability to provide uniform shared-memory access to a significant number of processors in a single SMP node brings us much closer to the ideal parallel computer envisioned over 20 years ago by theoreticians, the *Parallel Random Access Machine (PRAM)* (see, e.g., [1.44, 1.67]) and thus might enable us at long last to take advantage of 20 years of research in PRAM algorithms for various irregular computations. Moreover, as more and more supercomputers use the SMP cluster architecture, SMP computations will play a significant role in supercomputing as well.

1.8.1 Algorithms for SMPs

While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work. The number of processors remains quite low compared to the polynomial number of processors assumed by the PRAM model. This difference by itself would not pose a great problem: we can easily initiate far more processes or threads than we have processors. But we need

[2] `http://liinwww.ira.uka.de/~skampi/cgi-bin/run_list.cgi.pl`

algorithms with efficiency close to one and parallelism needs to be sufficiently coarse grained that thread scheduling overheads do not dominate the execution time. Another big difference is in synchronization and memory access: an SMP cannot support concurrent read to the same location by a thousand threads without significant slowdown and cannot support concurrent write at all (not even in the arbitrary CRCW model) because the unsynchronized writes could take place far too late to be used in the computation. In spite of these problems, SMPs provide much faster access to their shared-memory than an equivalent message-based architecture: even the largest SMP to date, the 106-processor "Starcat" Sun Fire E15000, has a memory access time of less than $300ns$ to its entire physical memory of 576GB, whereas the latency for access to the memory of another processor in a message-based architecture is measured in tens of microseconds—in other words, message-based architectures are 20–100 times slower than the largest SMPs in terms of their worst-case memory access times.

The Sun SMPs (the older "Starfire" [1.23] and the newer "Starcat") use a combination of large (16×16) data crossbar switches, multiple snooping buses, and sophisticated handling of local caches to achieve uniform memory access across the entire physical memory. However, there remains a large difference between the access time for an element in the local processor cache (below $5ns$ in a Starcat) and that for an element that must be obtained from memory (around $300ns$)—and that difference increases as the number of processors increases.

1.8.2 Leveraging PRAM Algorithms for SMPs

Since current SMP architectures differ significantly from the PRAM model, we need a methodology for mapping PRAM algorithms onto SMPs. In order to accomplish this mapping we face four main issues: (i) change of programming environment; (ii) move from synchronous to asynchronous execution mode; (iii) sharp reduction in the number of processors; and (iv) need for cache awareness. We now describe how each of these issues can be handled; using these approaches, we have obtained linear speedups for a collection of nontrivial combinatorial algorithms, demonstrating nearly perfect scaling with the problem size and with the number of processors (from 2 to 32) [1.6].

Programming Environment. A PRAM algorithm is described by pseudocode parameterized by the index of the processor. An SMP program must add to this explicit synchronization steps—software barriers must replace the implicit lockstep execution of PRAM programs. A friendly environment, however, should also provide primitives for memory management for shared-buffer allocation and release, as well as for contextualization (executing a statement on only a subset of processors) and for scheduling n independent work statements implicitly to $p < n$ processors as evenly as possible.

Synchronization. The mismatch between the lockstep execution of the PRAM and the asynchronous nature of parallel architecture mandates the use of software barriers. In the extreme, a barrier can be inserted after each PRAM step to guarantee a lock-step synchronization—at a high level, this is what the BSP model does. However, many of these barriers are not necessary: concurrent read operations can proceed asynchronously, as can expression evaluation on local variables. What needs to be synchronized is the writing to memory—so that the next read from memory will be consistent among the processors. Moreover, a concurrent write must be serialized (simulated); standard techniques have been developed for this purpose in the PRAM model and the same can be applied to the shared-memory environment, with the same $\log p$ slowdown.

Number of Processors. Since a PRAM algorithm may assume as many as $n^{O(1)}$ processors for an input of size n—or an arbitrary number of processors for each parallel step, we need to *schedule* the work on an SMP, which will always fall short of that resource goal. We can use the lower-level scheduling principle of the work-time framework [1.44] to schedule the $W(n)$ operations of the PRAM algorithm onto the fixed number p of processors of the SMP. In this way, for each parallel step k, $1 \leq k \leq T(n)$, the $W_k(n)$ operations are simulated in at most $W_k(n)/p + 1$ steps using p processors. If the PRAM algorithm has $T(n)$ parallel steps, our new schedule has complexity of $O\left(W(n)/p + T(n)\right)$ for any number p of processors. The work-time framework leaves much freedom as to the details of the scheduling, freedom that should be used by the programmer to maximize cache locality.

Cache-Awareness. SMP architectures typically have a deep memory hierarchy with multiple on-chip and off-chip caches, resulting currently in two orders of magnitude of difference between the best-case (pipelined preloaded cache read) and worst-case (non-cached shared-memory read) memory read times. A cache-aware algorithm must efficiently use both spatial and temporal locality in algorithms to optimize memory access time. While research into cache-aware sequential algorithms has seen early successes (see [1.54] for a review), the design for *multiple* processor SMPs has barely begun. In an SMP, the issues are magnified in that not only does the algorithm need to provide the best spatial and temporal locality to each processor, but the algorithm must also handle the system of processors and cache protocols. While some performance issues such as false sharing and granularity are well-known, no complete methodology exists for practical SMP algorithmic design. Optimistic preliminary results have been reported (e.g., [1.59, 1.63]) using OpenMP on an SGI Origin2000, cache-coherent non-uniform memory access (ccNUMA) architecture, that good performance can be achieved for several benchmark codes from NAS and SPEC through automatic data distribution.

1.9 Conclusions

Parallel computing is slowly emerging from its niche of specialized, expensive hardware and restricted applications to become part of everyday computing. As we build support libraries for desktop parallel computing or for newer environments such as large-scale shared-memory computing, we need tools to ensure that our library modules (or application programs built upon them) are as efficient as possible. Producing efficient implementations is the goal of algorithm engineering, which has demonstrated early successes in sequential computing. In this article, we have reviewed the new challenges to algorithm engineering posed by a parallel environment and indicated some of the approaches that may lead to solutions.

Acknowledgments

This work was supported in part by National Science Foundation grants CAREER ACI 00-93039 (Bader), ACI 00-81404 (Moret/Bader), DEB 99-10123 (Bader), EIA 01-21377 (Moret/Bader), EIA 01-13095 (Moret), and DEB 01-20709 (Moret/Bader), and by the Future and Emerging Technologies program of the EU under contract number IST-1999-14186 (Sanders).

References

1.1 A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

1.2 A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: iNcorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. In *Proceedings of the 7th Annual Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105, 1995.

1.3 E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Cros, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 2nd edition, 1995.

1.4 D. A. Bader. An improved randomized selection algorithm with an experimental study. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX'00)*, pages 115–129, 2000. `www.cs.unm.edu/Conferences/ALENEX00/`.

1.5 D. A. Bader, D. R. Helman, and J. JáJá. Practical parallel algorithms for personalized communication and integer sorting. *ACM Journal of Experimental Algorithmics*, 1(3):1–42, 1996. `www.jea.acm.org/1996/BaderPersonalized/`.

1.6 D. A. Bader, A. K. Illendula, B. M. E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*. Springer Lecture Notes in Computer Science 2141, pages 129–144, 2001.

1.7 D. A. Bader and J. JáJá. Parallel algorithms for image histogramming and connected components with an experimental study. *Journal of Parallel and Distributed Computing*, 35(2):173–190, 1996.

1.8 D. A. Bader and J. JáJá. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 292–301, 1996.
1.9 D. A. Bader and J. JáJá. SIMPLE: a methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.
1.10 D. A. Bader, J. JáJá, and R. Chellappa. Scalable data parallel algorithms for texture synthesis using Gibbs random fields. *IEEE Transactions on Image Processing*, 4(10):1456–1460, 1995.
1.11 D. A. Bader, J. JáJá, D. Harwood, and L. S. Davis. Parallel algorithms for image enhancement and segmentation by region growing with an experimental study. *Journal on Supercomputing*, 10(2):141–168, 1996.
1.12 D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, March 1994.
1.13 D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputer Review*, 4(8):54–55, 1991.
1.14 R. D. Barve and J. S. Vitter. A simple and efficient parallel disk mergesort. In *Proceedings of the 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 232–241, 1999.
1.15 A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, 203(2):175–203, 1998.
1.16 A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Priority queue operations and selection for the BSP* model. In *Proceedings of the 2nd International Euro-Par Conference*. Springer Lecture Notes in Computer Science 1124, pages 369–376, 1996.
1.17 A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic parallel algorithms: priority queue operations and selection for the BSP* model. In *Proceedings of the 2nd International Euro-Par Conference*. Springer Lecture Notes in Computer Science 1124, pages 27–29, 1996.
1.18 D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: a parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, vol. 1, pages 11–14, 1995.
1.19 L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.
1.20 G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16, 1991.
1.21 G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.
1.22 O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library — design, implementation and performance. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium Parallel and Distributed Processing (IPPS/SPDP'99)*, 1999. `www.uni-paderborn.de/~pub/`.

1.23 A. Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

1.24 J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computations*, pages 120–127, 1992.

1.25 D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Portability and Performance for Parallel Processing*, chapter 4, pages 71–98. John Wiley & Sons, 1993.

1.26 D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the 4th Symposium on the Principles and Practice of Parallel Programming*, pages 1–12, 1993.

1.27 J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V.W. Reynders, S. A. Smith, and T. J. Williams. Rapid application development and enhanced code interoperabily using the POOMA framework. In M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors, *Proceedings of the 1998 Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, chapter 29. SIAM, Yorktown Heights, NY, 1999.

1.28 P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proceedings of the 2nd International Euro-Par Conference*, pages 352–358, 1996.

1.29 S. J. Fink and S. B. Baden. Runtime support for multi-tier programming of block-structured applications on SMP clusters. In Y. Ishikawa et al., editors, *Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*. Springer Lecture Notes in Computer Science 1343, pages 1–8, 1997.

1.30 M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, 1998.

1.31 M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pages 285–297, 1999.

1.32 A. V. Goldberg and B. M. E. Moret. Combinatorial algorithms test sets (CATS): the ACM/EATCS platform for experimental research. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA'99)*, pages 913–914, 1999.

1.33 W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical report, Argonne National Laboratory, Argonne, IL, 1996. `www.mcs.anl.gov/mpi/mpich/`.

1.34 S. E. Hambrusch and A. A. Khokhar. C^3: a parallel model for coarse-grained machines. *Journal of Parallel and Distributed Computing*, 32:139–154, 1996.

1.35 D. R. Helman, D. A. Bader, and J. JáJá. A parallel sorting algorithm with an experimental study. Technical Report CS-TR-3549 and UMIACS-TR-95-102, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1995.

1.36 D. R. Helman, D. A. Bader, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 211–220, 1996.

1.37 D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.
1.38 D. R. Helman and J. JáJá. Sorting on clusters of SMP's. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS'98)*, pages 1–7, 1998.
1.39 D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experiments (ALENEX'98)*. Springer Lecture Notes in Computer Science 1619, pages 37–56, 1998.
1.40 D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.
1.41 D. R. Helman, J. JáJá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *ACM Journal of Experimental Algorithmics*, 3(4), 1997. `www.jea.acm.org/1998/HelmanSorting/`.
1.42 High Performance Fortran Forum. *High Performance Fortran Language Specification*, edition 1.0, May 1993.
1.43 J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory, 1997. `www.BSP-Worldwide.org/implmnts/oxtool/`.
1.44 J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992.
1.45 B. H. H. Juurlink and H. A. G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 13(3):271–318, 1998.
1.46 S. N. V. Kalluri, J. JáJá, D. A. Bader, Z. Zhang, J. R. G. Townshend, and H. Fallah-Adl. High performance computing algorithms for land cover dynamics using remote sensing data. *International Journal of Remote Sensing*, 21(6):1513–1536, 2000.
1.47 J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, 1997.
1.48 C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 199.
1.49 M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1998.
1.50 C. C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.
1.51 Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.
1.52 F. Meyer auf der Heide and R. Wanka. Parallel bridging models and their impact on algorithm design. In *Proceedings of the International Conference on Computational Science, Part II*, Springer Lecture Notes in Computer Science 2074, pages 628–637, 2001.
1.53 B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. *Journal on Supercomputing*, 22:99–111, 2002. Special issue on the best papers from *ICCS'01*.

1.54 B. M. E. Moret and H. D. Shapiro. Algorithms and experiments: the new (and old) methodology. *Journal of Universal Computer Science*, 7(5):434–446, 2001.
1.55 B. M. E. Moret, A. C. Siepel, J. Tang, and T. Liu. Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI'02)*. Springer Lecture Notes in Computer Science 2542, 2002.
1.56 MRJ Inc. The Portable Batch System (PBS). `pbs.mrj.com`.
1.57 F. Müller. A library implementation of POSIX threads under UNIX. In *Proceedings of the 1993 Winter USENIX Conference*, pages 29–41, 1993. `www.informatik.hu-berlin.de/~mueller/projects.html`.
1.58 W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: visualization and analysis of MPI resources. *Supercomputer 63*, 12(1):69–80, January 1996.
1.59 D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in OpenMP. In *Proceedings of Supercomputing*, 2000.
1.60 Ohio Supercomputer Center. *LAM/MPI Parallel Computing*. The Ohio State University, Columbus, OH, 1995. `www.lam-mpi.org`.
1.61 OpenMP Architecture Review Board. OpenMP: a proposed industry standard API for shared memory programming. `www.openmp.org`, October 1997.
1.62 Platform Computing Inc. The Load Sharing Facility (LSF). `www.platform.com`.
1.63 E. D. Polychronopoulos, D. S. Nikolopoulos, T. S. Papatheodorou, X. Martorell, J. Labarta, and N. Navarro. An efficient kernel-level scheduling methodology for multiprogrammed shared memory multiprocessors. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, 1999.
1.64 POSIX. *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*. Portable Applications Standards Committee of the IEEE, edition 1996-07-12, 1996. ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1.
1.65 N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX'00)*, pages 131–146, 2000. `www.cs.unm.edu/Conferences/ALENEX00/`.
1.66 D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, and L. F. Tavera. Scalable performance analysis: the Pablo performance analysis environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113, 1993.
1.67 J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.
1.68 R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: a detailed, accurate MPI benchmark. In *Proceedings of EuroPVM/MPI'98*. Springer Lecture Notes in Computer Science 1497, pages 52–59, 1998. See also `liinwww.ira.uka.de/~skampi/`.
1.69 R. Reussner, P. Sanders, and J. Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 2001. Accepted, conference version with L. Prechelt and M. Müller in *Proceedings of EuroPVM/MPI'98*.
1.70 P. Sanders. Load balancing algorithms for parallel depth first search (In German: Lastverteilungsalgorithmen für parallele Tiefensuche). Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, Berlin, 1997.

1.71 P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998. Special Issue on Parallel and Distributed Data Structures.
1.72 P. Sanders. Accessing multiple sequences through set associative caches. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*. Springer Lecture Notes in Computer Science 1644, pages 655–664, 1999.
1.73 P. Sanders and T. Hansch. On the efficient implementation of massively parallel quicksort. In *Proceedings of the 4th International Workshop on Solving Irregularly Structured Problems in Parallel (IRREGULAR'97)*. Springer Lecture Notes in Computer Science 1253, pages 13–24, 1997.
1.74 U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 410–414, 1999.
1.75 S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms (SODA'00)*, pages 829–838, 2000.
1.76 T. L. Sterling, J. Salmon, and D. J. Becker. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, Cambridge, MA, 1999.
1.77 L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
1.78 J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.
1.79 J. S. Vitter and E. A.M. Shriver. Algorithms for parallel memory II: hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.
1.80 R. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). In *Proceedings of Supercomputing'98*, 1998. `www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps`.
1.81 H. A. G. Wijshoff and B. H. H. Juurlink. A quantitative comparison of parallel computation models. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 13–24, 1996.
1.82 Y. Yan and X. Zhang. Lock bypassing: an efficient algorithm for concurrently accessing priority heaps. *ACM Journal of Experimental Algorithmics*, 3(3), 1998. `www.jea.acm.org/1998/YanLock/`.
1.83 Z. Zhang, J. JáJá, D. A. Bader, S. Kalluri, H. Song, N. El Saleous, E. Vermote, and J. Townshend. Kronos: A Software System for the Processing and Retrieval of Large-Scale AVHRR Data Sets. *Photogrammetric Engineering and Remote Sensing*, 66(9):1073–1082, September 2000.

1.A Examples of Algorithm Engineering for Parallel Computation

Within the scope of this paper, it would be difficult to provide meaningful and self-contained examples for each of the various points we made. In lieu of such target examples, we offer here several references[3] that exemplify the best aspects of algorithm engineering studies for high-performance and parallel

[3] We do not attempt to include all of the best work in the area: our selection is perforce idiosyncratic.

computing. For each paper or collection of papers, we describe those aspects of the work that led to its inclusion in this section.

1. The authors' prior publications [1.53, 1.6, 1.4, 1.46, 1.9, 1.71, 1.68, 1.37, 1.41, 1.73, 1.36, 1.5, 1.11, 1.8, 1.7, 1.10] contain many empirical studies of parallel algorithms for combinatorial problems like sorting [1.5, 1.35, 1.41, 1.73, 1.36], selection [1.4, 1.71, 1.8], and priority queues [1.71], graph algorithms [1.53], backtrack search [1.70], and image processing [1.46, 1.11, 1.7, 1.10].
2. JáJá and Helman conducted empirical studies for prefix computations [1.40], sorting [1.38] and list-ranking [1.39] on symmetric multiprocessors. The sorting paper [1.38] extends Vitter's external Parallel Disk Model [1.1, 1.78, 1.79] to the internal memory hierarchy of SMPs and uses this new computational model to analyze a general-purpose sample sort that operates efficiently in shared-memory. The performance evaluation uses 9 well-defined benchmarks. The benchmarks include input distributions commonly used for sorting benchmarks (such as keys selected uniformly and at random), but also benchmarks designed to challenge the implementation through load imbalance and memory contention and to circumvent algorithmic design choices based on specific input properties (such as data distribution, presence of duplicate keys, pre-sorted inputs, etc.).
3. In [1.20, 1.21] Blelloch *et al.* compare through analysis and implementation three sorting algorithms on the Thinking Machines CM-2. Despite the use of an outdated (and no longer available) platform, this paper is a gem and should be required reading for every parallel algorithm designer. In one of the first studies of its kind, the authors estimate running times of four of the machine's primitives, then analyze the steps of the three sorting algorithms in terms of these parameters. The experimental studies of the performance are normalized to provide clear comparison of how the algorithms scale with input size on a $32K$-processor CM-2.
4. Vitter *et al.* provide the canonical theoretic foundation for I/O-intensive experimental algorithmics using external parallel disks (e.g., see [1.1, 1.78, 1.79, 1.14]). Examples from sorting, FFT, permuting, and matrix transposition problems are used to demonstrate the parallel disk model. For instance, using this model in [1.14], empirical results are given for external sorting on a fixed number of disks with from 1 to 10 million items, and two algorithms are compared with overall time, number of merge passes, I/O streaming rates, using computers with different internal memory sizes.
5. Hambrusch and Khokhar present a model (C^3) for parallel computation that, for a given algorithm and target architecture, provides the complexity of computation, communication patterns, and potential communication congestion [1.34]. This paper is one of the first efforts to model collective communication both theoretically and through experiments, and then validate the model with coarse-grained computational

applications on an Intel supercomputer. Collective operations are thoroughly characterized by message size and higher-level patterns are then analyzed for communication and computation complexities in terms of these primitives.

6. While not itself an experimental paper, Meyer auf der Heide and Wanka demonstrate in [1.52] the impact of features of parallel computation models on the design of efficient parallel algorithms. The authors begin with an optimal multisearch algorithm for the Bulk Synchronous Parallel (BSP) model that is no longer optimal in realistic extensions of BSP that take critical blocksize into account such as BSP* (e.g., [1.17, 1.16, 1.15]). When blocksize is taken into account, the modified algorithm is optimal in BSP*. The authors present a similar example with a broadcast algorithm using a BSP model extension that measures locality of communication, called D-BSP [1.28].
7. Juurlink and Wijshoff [1.81, 1.45] perform one of the first detailed experimental accounts on the preciseness of several parallel computation models on five parallel platforms. The authors discuss the predictive capabilities of the models, compare the models to find out which allows for the design of the most efficient parallel algorithms, and experimentally compare the performance of algorithms designed with the model versus those designed with machine-specific characteristics in mind. The authors derive model parameters for each platform, analyses for a variety of algorithms (matrix multiplication, bitonic sort, sample sort, all-pairs shortest path), and detailed performance comparisons.
8. The LogP model of Culler *et al.* [1.26] (and its extensions such as logGP [1.2] for long messages) provides a realistic model for designing parallel algorithms for message-passing platforms. Its use is demonstrated for a number of problems, including sorting [1.25]. Four parallel sorting algorithms are analyzed for LogP and their performance on parallel platforms with from 32 to 512 processors is predicted by LogP using parameter values for the machine. The authors analyze both regular and irregular communication and provide normalized predicted and measured running times for the steps of each algorithm.
9. Yun and Zhang [1.82] describe an extensive performance evaluation of lock bypassing for concurrent access to priority heaps. The empirical study compares three algorithms by reporting the average number of locks waited for in heaps of 255 and 512 nodes. The average hold operation times are given for the three algorithms for uniform, exponential, and geometric, distributions, with inter-hold operation delays of 0, 160, and 640μs.
10. Several research groups have performed extensive algorithm engineering for high-performance numerical computing. One of the most prominent efforts is that led by Dongarra for ScaLAPACK [1.24, 1.19], a scalable linear algebra library for parallel computers. ScaLAPACK encapsulates

much of the high-performance algorithm engineering with significant impact to its users who require efficient parallel versions of matrix-matrix linear algebra routines. In [1.24], for instance, experimental results are given for parallel LU factorization plotted in performance achieved (gigaflops per second) for various matrix sizes, with a different series for each machine configuration. Because ScaLAPACK relies on fast sequential linear algebra routines (e.g., LAPACK [1.3]), new approaches for automatically tuning the sequential library (e.g., LAPACK) are now available as the ATLAS package [1.80].

2. Visualization in Algorithm Engineering: Tools and Techniques*

Camil Demetrescu[1], Irene Finocchi[2],
Giuseppe F. Italiano[3], and Stefan Näher[4]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Rome, Italy
demetres@dis.uniroma1.it
[2] Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza", Rome, Italy
finocchi@dsi.uniroma1.it
[3] Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Rome, Italy
italiano@info.uniroma2.it
[4] FB IV — Informatik, Universität Trier, Germany
naeher@informatik.uni-trier.de

Summary.

The process of implementing, debugging, testing, engineering and experimentally analyzing algorithmic codes is a complex and delicate task, fraught with many difficulties and pitfalls. In this context, traditional low-level textual debuggers or industrial-strength development environments can be of little help for algorithm engineers, who are mainly interested in high-level algorithmic ideas and not particularly in the language and platform-dependent details of actual implementations. Algorithm visualization environments provide tools for abstracting irrelevant program details and for conveying into still or animated images the high-level algorithmic behavior of a piece of software.

In this paper we address the role of visualization in algorithm engineering. We survey the main approaches and existing tools and we discuss difficulties and relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes.

2.1 Introduction

There has been an increasing attention in our community toward the experimental evaluation of algorithms. Indeed, several tools whose target is to offer a general-purpose workbench for the experimental validation and fine-tuning of algorithms and data structures have been produced: software repositories

* This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT), by CNR, the Italian National Research Council under contract n. 00.00346.CT26, and by DFG-Grant Na 303/1-2, Forschungsschwerpunkt "Effiziente Algorithmen für diskreteProbleme und ihre Anwendungen".

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 24–50, 2002.
© Springer-Verlag Berlin Heidelberg 2002

and libraries, collections and generators of test sets, software systems for supporting implementation and analysis are relevant examples of this effort. In particular, in the last years there has been increasing attention toward the design and implementation of interactive environments for developing and experimenting with algorithms, such as editors for test sets and development, debugging, and visualization tools.

In this paper we address the role of algorithm visualization tools in algorithm engineering. According to a standard definition [2.44], algorithm animation is a form of high-level dynamic software visualization that uses graphics and animation techniques for portraying and monitoring the computational steps of algorithms. Systems for algorithm animation have matured significantly since, in the last decade, high-quality user interfaces have become a standard in a large number of areas.

Nevertheless, the birth of algorithm visualization can be dated back to the 60's, when Licklider did early experiments on the use of graphics for monitoring the evolution of the content of a computer memory. Knowlton was the first to address the visualization of dynamically changing data structures in his films demonstrating the Bell Lab's low-level list processing language [2.29]. During the 70's, the potential of program animation in a pedagogical setting was pointed out by several authors, and this research ended up with the realization in 1981 of the videotape Sorting Out Sorting [2.3], which represents a milestone in the history of algorithm animation and has been successfully used to teach sorting methods to computer science students for more than 15 years. Sorting Out Sorting is a 30-minute color film that explains nine internal sorting algorithms, illustrating both their substance and their differences in efficiency. Different graphical representations are provided for the data being sorted, and showing the programs while running on their input makes it clear at any step how such data are partially reorganized by the different algorithms. A new era began with the 80's, when bit-mapped displays became available on workstations: researchers attempted to go beyond films and started developing interactive software visualization systems and exploring their utility not only for education, but also for software engineering and program debugging. Dozens of algorithm animation systems have been developed since then.

Thanks to the capability of conveying a large amount of information in a compact form and to the ability of human beings at processing visual information, algorithm animation systems are useful tools also in algorithm engineering, in particular in several phases during the process of design, implementation, analysis, tuning, experimental evaluation, and presentation of algorithms. Actually, visual debugging techniques can help highlight hidden programming or conceptual errors, i.e., discover both errors due to a wrong implementation of an algorithm and, at a higher level of abstraction, errors possibly due to an incorrect design of the algorithm itself. Sometimes, algorithm animation tools can help in designing heuristics and local improvements

in the code difficult to figure out theoretically, to test the correctness of algorithms on specific test sets, to discover degeneracies, i.e., special cases for which the algorithm may not produce a correct output. Their use can leverage the task of monitoring complex systems or complex programs (e.g., concurrent programs), and makes it possible also to analyze problem instances not limited in size and complexity, which even long and boring handiwork would not be able to deal with. Not last, visualization could be an attractive medium for algorithms researchers who want to share and disseminate their ideas.

In spite of the great research devoted in recent years to designing and developing algorithm visualization facilities, the diffusion of the use of such systems for algorithm engineering is still limited. We believe this is mostly due to the lack of fast prototyping mechanisms, i.e., to the fact that realizing an animation often requires heavy modifications of the source code at hand and therefore a great effort. Instead, the power of an algorithm animation system should be in the hands of the end-users, possibly unexperienced, rather than of professional programmers or of the developers of the visualization tool. In addition, it is very important for a software visualization tool to be able to animate not just "toy programs", but significantly complex algorithmic codes, and to test their behavior on large data sets. Unfortunately, even those systems well suited for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck, such as changes of resolution and scale, selectivity, and elision of information. Finding a solution to this kind of limitations is nowadays a challenge for algorithm visualization systems.

In this paper we survey the main approaches and existing tools for the realization of animations of algorithms. In particular, Section 2.2 is concerned with the description of software visualization systems and libraries supporting visualization capabilities. Section 2.3 describes two main approaches used by visualization tools: interesting events and state mapping. In Section 2.4 we discuss difficulties and present relevant examples where visualization systems helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. Conclusions and challenges for algorithm visualization research are finally listed in Section 2.5.

2.2 Tools for Algorithm Visualization

In this section we survey some algorithm visualization systems, discussing their main features and the different approaches introduced by each of them. We do not aim at being exhaustive, but rather we try to highlight the aspects of these systems interesting from an algorithm engineering point of view. We also describe some tools that will be used in Section 2.4 for illustrating how to prepare algorithm animations for debugging or demonstrations. We attempt to present visualization systems by their focus and innovation. For

a more comprehensive description of software visualization systems we refer the interested reader to [2.44] and to the references therein.

Balsa [2.8], followed a few years later by Balsa-II [2.9], was the first system able to animate general-purpose algorithms and pioneered the interesting events approach, later used by many other tools. In order to realize an animation, the points of the source code that are strategically important are annotated with procedure calls that generate visualization events. At run time, events are collected by an event manager that forwards them to the views so as to update the displayed images. Balsa-II supports a good level of interactivity, allowing execution control by means of step-points and stop-points. In order to provide a measure of an algorithm's performance, it also supports a way to associate different costs to different events and to count the number of times each interesting event occurs, which may be interesting for profiling algorithmic codes. Zeus [2.11] is an evolution of Balsa-II and adds to the interesting events approach some object-oriented features: each view is created by deriving a standard base `View` class and can be provided with additional methods to handle each interesting event. Zeus also extensively uses color and sound [2.12] and deals with three-dimensional objects [2.13], thus making it possible to realize highly-customizable visualizations.

TANGO [2.42] (Transition-based ANimation GeneratiOn) introduced the path-transition paradigm [2.41] for creating smooth continuous animations. This paradigm relies on the use of four abstract data types (location, image, path, and transition) and animations are realized by handling instances of these data types by means of suitable operations defined on them. X-TANGO [2.43] is the X-Windows based follow-up of TANGO. Polka [2.45] introduces the support for the animation of concurrent programs: the programmer can assemble and present the whole animation using an explicit global clock counter as a timing system. It also includes a graphical front-end, called Samba, that is driven by a script produced as a trace of the execution.

Debugging concurrent programs is more complicated than understanding the behavior of sequential codes: not only concurrent computations may produce vast quantities of data, but also the presence of multiple threads that communicate, compete for resources, and periodically synchronize may result in unexpected interactions and non-deterministic executions. Many tools have been realized to cope with these issues. The Gthreads library [2.50] builds and displays a program graph as threads are forked and functions are called: the vertices represent program entities and events and the arcs temporal orderings between them. The Hence system [2.6] offers animated views of the program graph obtained from execution of PVM programs. Message passing views are supported by the Conch system [2.49]: processes appear around the outside of a ring and messages move from the sending process to the receiving one by traversing the center of the ring. This is useful to detect undelivered messages, as they remain in the center of the ring. Kiviat graphs

for monitoring the CPU utilization of each processor are also supported by other systems such as ParaGraph [2.25] and Tapestry [2.33].

One of the few examples of attempts to provide automatic visualization of simple data structures is UWPI [2.26] (University of Washington Program Illustrator). The visualization is automatically performed by the system thanks to an "inferencer" that analyzes the data structures in the source code, both at compile-time and at run-time, and suggests a number of possible displays for each of them. Clearly, due to the lack of a deep knowledge of the logic of the program, only the visualization of simple data structures, such as stacks or queues, can be supported.

Pavane [2.38, 2.40] marks the first paradigm shift in algorithm visualization since the introduction of interesting events. It features a declarative approach to the visualization of concurrent programs. It conceives the visualization as a mapping between the state of the computation and the image space: the transformation between a fixed set of program variables and the final image is declared by using suitable rules. This seems very important for developing visual debugging tools for languages such as Prolog and Lisp. Furthemore, the non-invasiveness of the declarative approach seems very important also in a concurrent framework, since the execution may be nondeterministic and an invasive visualization code may change the outcome of a computation.

TPM [2.21] (Transparent Prolog Machine) is a debugging tool for the post-mortem visualization of computer programs written in the Prolog programming language. In order to deal with the inherent complexity of Prolog programs, TPM features two distinct views: a fine-grained view to represent the program's locality and a coarse-grained view to show the full execution space via animated AND-OR trees. The overall trace structure also captures the concept of backtracking to find alternative solutions to goals. ZStep95 [2.32] is a reversible and animated source code stepper for LISP programs that provides a powerful mechanism for error localization. It maintains a complete history of the execution and is equipped with a fully reversible control structure: the user allows the program to run until an error is found and then can go back to discover the exact point in which something went wrong. Moreover, a simple and strict connection between the execution and its graphical output is obtained by elementary clicking actions.

Leonardo [2.17] is an integrated environment for developing, animating, and executing general-purpose C programs. Animations are realized according to a declarative approach, i.e., by embedding in the source code declarations that provide high-level graphical interpretations of the program's variables. As the system automatically reflects the modifications of the program state into the displayed images, a high level of automation is reached. Animations can be fully customized by means of a graphical vocabulary including basic geometric shapes as well as primitives for visualizing graphs and trees. Smoothly changing images are also supported by the system to help

the viewer maintain context [2.19]. In addition, code written with Leonardo is completely reversible: when running code backwards, variable assignments are undone, output sent to the console disappears, graphics drawn are undrawn, and so on. The reversibility is extended to the full set of standard ANSI functions. This feature, combined with the declarative approach, makes the system well suited for visual debugging purposes. Differently from many other visualization systems, Leonardo has been widely distributed over the Internet and includes several animations of algorithms and data structures.

Computational geometry is an area where the visualization and animation of programs is a very important tool for the understanding, presentation, and debugging of algorithms, and the animation of geometric algorithms is mentioned among the strategic research directions in computational geometry [2.47]. It is thus not surprising that increasing attention has been devoted to algorithm visualization tools for computational geometry (see, e.g., [2.2, 2.4, 2.20, 2.27, 2.46]). In this paper we particularly focus our attention on *GeoWin*, a `C++` data type that can be easily interfaced with algorithmic software libraries of great importance in algorithm engineering such as CGAL [2.22] and LEDA [2.34]. The design and implementation of GeoWin was influenced by LEDA's graph editor *GraphWin* (see [2.34], Chapter 12). Both data types support a number of programming styles that have proven to be useful in demonstration and animation programs. Examples are the use of *result scenes* and the *event handling* approach, which will be discussed in Section 2.4.3. An instance *gw* of the data type $GeoWin$ is an editor that maintains a collection of so-called *scenes*. Each scene in this collection has an associated *container* of geometric *objects* whose members are displayed according to a set of visual attributes (color, line width, line style, etc.). One of the scenes in the collection can be *active*. It receives the input of all editing operations and can be manipulated through the interactive interface. Both the container type and the object type have to provide a certain functionality. The container type must implement the STL list interface [2.35], in particular, it has to provide STL-style iterators, and for all geometric objects a small number of functions and operators (for stream input and output, basic transformations, drawing and mouse input in a LEDA window) have to be defined. Any combination of container and object type that fulfill these requirements for containers and objects, respectively, can be associated with a GeoWin scene in a `gw.new_scene()` operation. More recent work on geometric visualization include VEGA [2.27] and WAVE [2.20].

VEGA (Visualization Environment for Geometric Algorithms) is a client-server visualization environment for geometric algorithms. It guarantees a low usage of communication bandwidth resources, thus achieving good performance even in slow networks. The end-user can interactively draw, load, save, and modify graphical scenes, can animate algorithms on-line or show saved runs off-line, and can customize the visualization by specifying a suitable set of view attributes. WAVE (Web Algorithm Visualization Engine) uses

a publication-driven approach to algorithm visualization over the Web and is especially well-suited for geometric algorithms. Algorithms run on a developer's remote server and their data structures are published on blackboards held by the clients. Animations are realized by writing suitable visualization handlers and by attaching them to the public data structures.

More recent trends in algorithm animation include distributed systems over the Web. JEliot [2.24, 2.31] is an automatic system for the animation of simple Java programs. After the Java code has been parsed, the user can choose the cast of variables to be visualized on the scene according to built-in graphical interpretations. The user needs to write no additional code: in other words, animation is embedded in the implementation of data type operations. The graphical presentation is based on a "theater metaphor" where the script is the algorithm, the stages are the views, the actors are the program's data structures depicted as graphical objects, and the director is the user.

CATAI [2.14] (Concurrent Algorithms and data Types Animation over the Internet) tries to minimize the burden of the task of animating algorithms. The main philosophy behind this system is that any algorithm implemented in an object-oriented programming language (such as `C++`) should be easily animated. This should make this system easy to use, and is based on the idea that an average programmer or an algorithm developer should not invest too much time in getting an animation of the algorithm up and running. This is not always the case, and often animating an algorithm can be as difficult and as time consuming as implementing the algorithm itself from scratch. CATAI has an advantage over systems where the task of animating an algorithm can be quite complex. Producing animations almost automatically, however, can limit flexibility in creating custom graphic displays. If the user is willing to invest more time on the development of an animation, he or she can produce more sophisticated graphics capabilities, while still exploiting the features offered by the system.

JDSL [2.5] (Java Data Structures Library) is a library of data structures written in the Java programming language that supports the visualization of the fundamental operations on abstract data types; it is well suited for educational purposes, as students obtain a predefined visualization of their own implementation by simply implementing JDSL Java interfaces with predefined signatures.

2.3 Interesting Events versus State Mapping

In this section we focus on the main features of animation systems that are appealing for their deployment in algorithm engineering. From the viewpoint of the algorithmic developer, it would be highly desirable to rely on systems that offer visualizations at a very high level of abstraction. Namely, one would be more interested in visualizing the behavior of a complex data structure,

such as a graph, than in obtaining a particular value of a given pointer. Furthermore, algorithm designers could be very interested in visualizations that are reusable and that can be created with little effort from the algorithmic source code at hand: this could be of substantial help in speeding up the time required to produce a running animation.

Achieving simultaneously high level of abstraction and fast prototyping makes the task of developing algorithm animation systems highly nontrivial. Indeed, it is possible to visualize automatically static or even running code, but at a very low level of abstraction, i.e., when the entities to be displayed and the way they change can be directly deduced from the code and the program state. For instance, the program counter tells us the next instruction, from which the line of the code to be executed can be easily recovered and highlighted in a suitable view. Also, primitive and composite data types can be mapped into canonical representations, thus displaying for free the data and the data flow. Conventional debuggers rely on this assumption but they lack capability of abstraction: they are unable to convey information about the algorithm's fundamental operations and to produce high-level synthesized views of data and of their manipulations. For example, if a graph is represented by means of an adjacency matrix, a debugger can automatically display only the matrix, but not the graph according to its usual representation with vertices and edges. Toward this aim, some extra knowledge, such as the interpretation of matrix entries, should be provided to the visualization system.

The considerations above are at the base of the distinction between *program* and *algorithm* visualization. In particular, an algorithm visualization system should be able to illustrate salient features of the algorithm, which appears to be difficult, if not impossible, with a completely automatic mechanism. The opposition *automation* versus *high-level* and *customization* possibilities makes it necessary to define a method for specifying the visualization, i.e., a suitable mechanism for binding pictures to code. In the remainder of this section, we discuss the two major solutions proposed in the literature: interesting events and state mapping. For a comprehensive discussion of other techniques used in algorithm visualization we refer the interested reader to [2.10, 2.36, 2.37, 2.39, 2.44].

Interesting Events. A natural approach to algorithm animation consists of annotating the algorithmic code with calls to visualization routines. The first step consists of identifying the relevant actions performed by the algorithm that are interesting for visualization purposes. Such relevant actions are usually referred to as *Interesting Events*. As an example, in a sorting algorithm the swap of two items can be considered an interesting event. The second step consists of associating each interesting event with a modification of a graphical scene. In our example, if we depict the values to be sorted as a sequence of sticks of different heights, the animation of a swap event might be realized by exchanging the positions of the two sticks corresponding to

the values being swapped. Animation scenes can be specified by setting up suitable visualization procedures that drive the graphic system according to the actual parameters generated by the particular event. Alternatively, these visualization procedures may simply log the events in a file for a *post-mortem* visualization. The calls to the visualization routines are usually obtained by annotating the original algorithmic code at the points where the interesting events take place. This can be done either by hand or by means of specialized editors.

In addition to being simple to implement, the main benefit of the event-driven approach is that interesting events are not necessarily low-level operations (such as comparisons or memory assignments), but can be more abstract and complex operations designed by the programmer and strictly related to the algorithm being visualized (e.g., the swap in the previous example, as well as a rotate operation in the management of an AVL tree). Major drawbacks include the following: realizing an animation may require the programmer to write several lines of additional code; the event-driven approach is invasive (even if the code is not transformed, it is augmented); the person who is in charge of realizing the animation has to know the source code quite well in order to identify all the interesting points.

State Mapping. Algorithm visualization systems based on state mapping rely on the assumption that observing how the variables change provides clues to the actions performed by the algorithm. The focus is on capturing and monitoring the data modifications rather than on processing the interesting events issued by the annotated algorithmic code. For this reason they are also referred to as "data driven" visualization systems. Conventional debuggers can be viewed as data driven systems, since they provide direct feedback of variable modifications.

Specifying an animation in a data driven system consists of providing a graphical interpretation of the *interesting data structures* of the algorithmic code. It is up to the system to ensure that the graphical interpretation at all times reflects the state of the computation of the program being animated. In the case of conventional debuggers, the interpretation is fixed and cannot be changed by the user: usually, a direct representation of the content of variables is provided. The debugger just updates the display after each change, sometimes highlighting the latest variable that has been modified by the program to help the user maintain context. In a more general scenario, an adjacency matrix used by the code may be visualized as a graph with vertices and edges, an array of numbers as a sequence of sticks of different heights, and a heap vector as a balanced tree. As the focus is only on data structures, the same graphical interpretation, and thus the same visualization code, may be reused for any algorithm that uses the same data structure. For instance, any sorting algorithm that manages to reorganize a given array of numbers may be animated with the same visualization code that displays the array as a sequence of sticks.

The main advantage of this approach over the event driven technique is that a much greater ignorance of the code is allowed: indeed, only the interpretation of the variables has to be known to animate a program. In Section 2.4.2 we will describe how we realized the animation of an algorithm in the system Leonardo with very little knowledge of the code to be visualized. On the other hand, focusing only on data modification may sometimes limit customization possibilities making it difficult to realize animations that would be natural to express with interesting events.

We believe that a combination of the two approaches described in this section would be most effective in algorithm animation as the two approaches capture different aspects of the problem. In our own experience, each of the two approaches has cases in which it is much preferable to the other. In some cases, we even found it useful to use both approaches simultaneously for realizing the same animation.

2.4 Visualization in Algorithm Engineering

In this section we present relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. In particular, we will consider examples where visualization can provide some insight into the design of algorithms at the level of profiling and experimental evaluation (Section 2.4.1) and where animation has greatly simplified the task of debugging complex algorithmic code (Section 2.4.2). One of the most important aspects of algorithm engineering is the development of libraries. It is thus quite natural to try to interface visualization tools to algorithmic software libraries. Two examples of such an effort are considered in Sections 2.4.3 and 2.4.4. In particular, we will show how demonstrations of geometric algorithms can be easily realized and interfaced with libraries (Section 2.4.3), and how fast animation prototyping can be achieved by reusing existing visualization code (Section 2.4.4).

2.4.1 Animation Systems and Heuristics: Max Flow

The maximum flow problem, first introduced by Berge and Ghouila-Houri in [2.7], is a fundamental problem in combinatorial optimization that arises in many practical applications. Examples of the maximum flow problem include determining the maximum steady-state flow of petroleum products in a pipeline network, cars in a road network, messages in a telecommunication network, and electricity in an electrical network. Given a capacitated network $G = (V, E, c)$ where V is the set of nodes, E is the set of edges and c_{xy} is the *capacity* of edge $(x, y) \in E$, the maximum flow problem consists of computing the maximum amount of flow that can be sent from a given

source node s to a given sink node t without exceeding the edge capacities. A flow assignment is a function f on edges such that $f_{xy} \leq c_{xy}$, i.e., edge capacities are not exceeded, and for each node v (except the source s and the sink t), $\sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw}$, i.e., the assigned incoming flows and the outgoing flows are equal. Usually, one needs to compute not only the maximum amount of flow that can be sent from the source to the sink in a given network, but also a flow assignment that achieves that amount.

Several methods for computing a maximum flow have been proposed in the literature. In particular, we mention the network simplex method proposed by Dantzig [2.18], the augmenting path method of Ford and Fulkerson, the blocking flow of Dinitz, and the push-relabel technique of Goldberg and Tarjan [2.1].

The push-relabel method, which made it possible to design the fastest algorithms for the maximum flow problem, sends flows locally on individual edges (push operation), possibly creating flow excesses at nodes, i.e., a *preflow*. A preflow is just a relaxed flow assignment such that for some nodes, called *active nodes*, the incoming flow may exceed the outgoing flow. The push-relabel algorithms work by progressively transforming the preflow into a maximum flow by dissipating excesses of flow held by active nodes that either reach the sink or return back to the source. This is done by repeatedly selecting a current active node according to some selection strategy, pushing as much excess flow as possible towards adjacent nodes that have a lower estimated distance from the sink paying attention not to exceed the edge capacities, and then, if the current node is still active, updating its estimated distance from the sink (relabel operation). Whenever an active node cannot reach the sink anymore as no path to the sink remains with some residual unused capacity, its distance progressively increases due to relabel operations until it gets greater than n: when this happens, it starts sending flow back towards the source, whose estimated distance is initially forced to n. This elegant solution makes it possible to deal with both sending flows to the sink and draining undeliverable excesses back to the source through exactly the same push/relabel operations. However, as we will see later, if taken "as is" this solution is not so good in practice.

Two aspects of the push-relabel technique seem to be relevant with respect to the running time: (1) the selection strategy of the current active node, and (2) the way estimated distances from the sink are updated by the algorithm.

The selection strategy of the current active node has been proved to significantly affect the asymptotic worst-case running time of push-relabel algorithms [2.1]: as a matter of fact, if active nodes are stored in a queue, the algorithm, usually referred to as the FIFO preflow-push algorithm, takes $O(n^3)$ in the worst case; if active nodes are kept in a priority queue where each extracted node has the maximum estimated distance from the sink, the worst-case running time decreases to $O(\sqrt{m}n^2)$, which is much better for

sparse graphs. The last algorithm is known as the highest-level preflow-push algorithm.

Unfortunately, regardless of the selection strategy, the push-relabel method in practice yields very slow codes if taken literally. Indeed, the way estimated distances from the sink are maintained has been proved to affect dramatically the practical performance of the push-relabel algorithms. For this reason, several additional heuristics for the problem have been proposed. Though these heuristics are irrelevant from an asymptotic point of view, the experimental study presented in [2.16] proves that two of them, i.e., the global relabeling and the gap heuristics, could be extremely useful in practice.

Global Relabeling Heuristic. Each relabel operation increases the estimated distance of the current active node from the sink to be equal to the lowest estimated distance of any adjacent node, plus one. This is done by considering only adjacent nodes joined by edges with some non-zero residual capacity, i.e., edges that can still carry some additional flows. As relabel operations are indeed local operations, the estimated distances from the sink may progressively deviate from the exact distances by losing the "big picture" of the distances: for this reason, flow excesses might not be correctly pushed right ahead towards the sink, and may follow longer paths slowing down the computation. The *global relabeling heuristic* consists of recomputing, say every n push/relabel operations, the exact distances from the sink, and the asymptotic cost of doing so can be amortized against the previous operations. This heuristic drastically improves the practical running time of algorithms based on the push-relabel method [2.16].

Gap Heuristic. Cherkassky [2.15] has observed that, at any time during the execution of the algorithm, if there are nodes with estimated distances from the sink that are strictly greater than some distance d and no other node has estimated distance d, then a gap in the distances has been formed and all active nodes above the gap will eventually send their flow excesses back to the source as they no longer can reach the sink. This can be achieved by repeatedly increasing the estimated distances via relabel operations. The process stops when distances get greater than n. The problem is that a huge number of such relabeling operations may be required. To avoid this, it is possible to keep track of gaps in the distances efficiently: whenever a gap occurs, the estimated distances of all nodes above the gap are immediately increased to n. This is usually referred to as the *gap heuristic* and, according to the study in [2.16], it is a very useful addition to the global relabeling heuristic if the highest-level active node selection strategy is applied. However, the gap heuristic does not seem to yield the same improvements under FIFO selection strategy.

The 5 snapshots a, b, c, d and e shown in Fig. 2.1 and in Fig. 2.2 have been produced by the algorithm animation system Leonardo [2.17] and depict the behavior of the highest-level preflow push algorithm implemented with

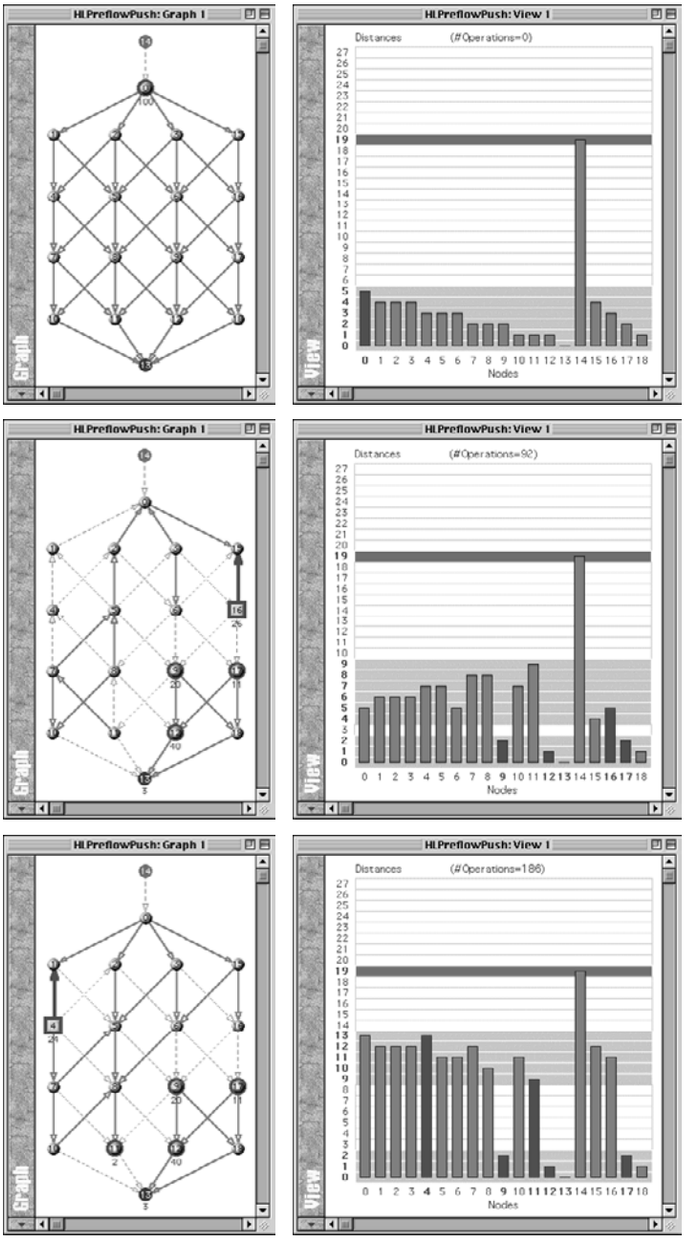
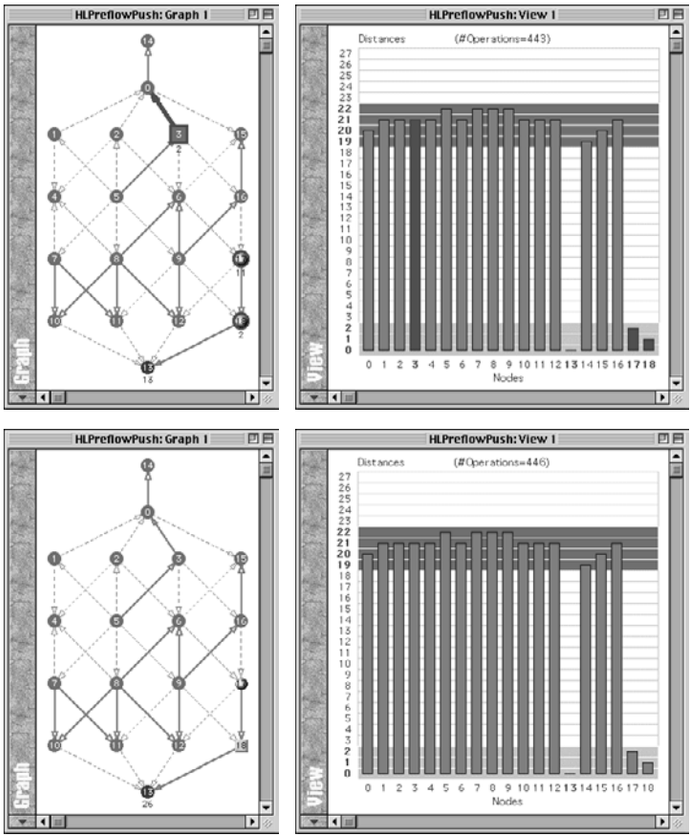

HLPreflowPush: Graph 1
HLPreflowPush: View 1
Distances
(#Operations=0)
Nodes
(a)
Network status and distances after the initialization phase.
HLPreflowPush: Graph 1
HLPreflowPush: View 1
Distances
(#Operations=92)
Nodes
(b)
After 92 operations a gap has been formed. Nodes with distance greater than the gap no longer can reach the sink. Their distance should be directly increased to n through the gap heuristic.
HLPreflowPush: Graph 1
HLPreflowPush: View 1
Distances
(#Operations=186)
Nodes
(c)
Nodes with distance greater than the gap are being slowly relabeled step after step if the gap heuristic is not implemented.

Fig. 2.1. Highest-level preflow push maxflow algorithm animation in Leonardo. Snapshots a, b, c



(d)

After 443 operations the distances of all nodes above the gap have been increased to n and their flow excesses are being drained back to the source. The gap heuristic could have saved the last 351 operations on this instance, i.e., about 80% of the total time spent by the algorithm to solve the problem.

(e)

After 446 operations the maximum flow has been determined by the algorithm and no more active nodes remain.

Fig. 2.2. Highest-level preflow push maxflow algorithm animation in Leonardo. Snapshots d, e

no additional heuristics on a small network with 19 nodes and 39 edges. The animation aims at giving an empirical explanation about the utility of the gap heuristic under the highest-level selection. The example shows that this heuristic, if added to the code, could have saved about 80% of the total time spent by the algorithm to solve the problem on that instance. Both the network and a histogram of the estimated distances of nodes are shown in the snapshots: active nodes are highlighted both in the network and in the histogram and flow excesses are reported as node labels. Moreover, the edge currently selected for a push operation is highlighted as well. Notice that the source is initially assigned distance n and all nodes that eventually send flows back to the source get distance greater than n. We believe that this is an example where a visualization system may be of great help in providing a

meaningful interpretation of data and statistics that can be of large size and intrinsically complex and heterogeneous.

To conclude this section, we briefly describe how this particular visualization was achieved with Leonardo. The source code used the following data type for representing the network:

```
struct network {
    int  n,s,t;            // Number of nodes, source and sink
    int  d[MAX];           // Estimated distances
    int  e[MAX];           // Flow excesses
    int  r[MAX][MAX];      // Residual capacity
    char adj[MAX][MAX];    // Boolean adjacency matrix
} G;                       // Instance of network data type
```

where `G` is an instance of the input network, with `G.n` nodes, source `G.s`, sink `G.t`, and Boolean adjacency matrix `G.adj[][]`. The algorithm maintains the estimated distances in `G.d[]`, the excess flow in `G.e[]`, and the residual capacities in `G.r[][]`. In order to produce the network visualization we embedded into the source code the following lines:

```
/**
   Graph(Out 1);
   Directed(1);
   Node(Out N,1) For N:InRange(N,0,G.n-1);
   Arc(X,Y,1)    If G.adj[X][Y]!=0;

   NodeColor(N,Out LightGreen,1) If G.d[N]>=G.n;
   NodeFrame(N,Out Red,Out 2,1)  If G.e[N]>0;
   NodeLabel(N,Out Int,Out L,1)  If G.e[N]>0 Assign L=G.e[N];

   ArcThickness(X,Y,Out Thick,1) If G.d[Y]==G.d[X]-1 &&  G.r[X][Y]>0;
   ArcStyle(X,Y,Out Dashed,1)    If G.d[Y]!=G.d[X]-1 || !G.r[X][Y];
**/
```

The goal of this visualization code is to declare a directed graph window displaying a graph with id number 1. The nodes of the visualized graph are in the range $[0, \texttt{G.n} - 1]$ and there is an edge $(\texttt{X}, \texttt{Y})$ if and only if the corresponding entry in the adjacency matrix is non-zero. Declarations of `NodeColor`, `NodeFrame`, `NodeLabel`, `NodeLabel`, `ArcStyle` and `ArcThickness` specify the graphical attributes of nodes and edges in the visualization. In particular, a node is colored light green if its estimated distance fron the sink is at least n; active nodes, i.e., nodes with positive excess flow, are highlighted with a red frame and the amount of integer (`Int`) excess flow is shown as a node label. Finally, edges are solid and thick if they might be selected for a push operation, i.e., they enter nodes with lower estimated distance from the sink and have positive residual capacity. The remaining edges are dashed. The animation hinges upon the fact that, when the original algorithmic code is executed, any change in the fields of variable `G` is automatically reflected in the displayed images. The visualization code for the window showing the estimated distances of nodes from the sink is based on similar ideas and not reported here.

2.4.2 Animation Systems and Debugging: Spring Embedding

In this section we address an important application of animation systems: debugging complex algorithmic codes. In particular, we describe our own experience with a graph layout algorithm and show how its animation was crucial for debugging an available implementation, and for discovering convergence problems due to numerical errors. The algorithm, due to Kamada and Kawai [2.28], is based on a force-directed paradigm, which uses a physical analogy to draw graphs: graph vertices and edges define a force system and the algorithm seeks a configuration with locally minimal energy. The embedding produced by the algorithm is also known as *spring embedding*: indeed, Kamada and Kawai's algorithm finds an embedding by introducing a force system where vertices are mutually connected by springs. In more detail, the algorithm attempts to find an embedding of the graph in which Euclidean distances between vertices are as close as possible to the lengths of their shortest paths. The energy of this system is thus:

$$E = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{k_{i,j}}{2} \left(dist(i,j) - L \cdot \ell(i,j) \right)^2 ,$$

where $dist(i,j)$ is the Euclidean distance between vertices i and j, $\ell(i,j)$ is the length of a shortest path between i and j in the embedded graph, L is the desirable length of a single edge in the drawing, and $k_{i,j}$ is the strength of the spring between vertices i and j.

In order to find a local minimum of the energy, Kamada and Kawai make use of a two-dimensional Newton-Raphson method to look where partial derivatives are zero (or close to zero). In particular, at each step all vertices are frozen, except for one vertex that is moved to a stable point by means of an iterated process. In more detail, the vertex with largest first-order partial derivatives is selected, and it is repeatedly moved towards a local minimum (based on the value of second-order partial derivatives). Those iterations terminate when the first-order partial derivatives become small enough. This is a very high-level description of the algorithm, which should suffice for our goals: the interested reader is referred to [2.28] for the full details of the method.

We received a C implementation of this algorithm that was implemented straight from the paper. The implementation seemed flawed with convergence problems on some graph instances: however, despite many efforts, the authors of the code were unable to track down the bug. We were thus asked to try to animate their implementation, in order to gain better understanding and help debug this piece of algorithmic code. At that time, we did not know much about Kamada and Kawai's algorithm, and did not know much about the implementation either: furthermore, we did not want to invest too much time in studying in depth either the paper or the implementation.

We set up an animation in Leonardo [2.17]: the only information we had to retrieve from the implementation concerned the data structures used to

store the graph and the positions of the vertices as progressively refined and returned by the algorithm. In particular, we had to look only at the following lines from the implementation code:

```
int n;
int G[ MAXNODES ][ MAXNODES ];
struct  { double x,y; } pos[ MAXNODES ];
```

where `G` is the adjacency matrix of the graph, `n` is the number of vertices, and `pos` contains the x and y coordinates of each vertex in the drawing. Our animation was set up so as to show how vertices were changing their position as the `pos` array was being updated by the algorithm, thus illustrating the intermediate drawings produced in different steps of the algorithm. We emphasize that it was very difficult to figure out this process using only the numerical information displayed by a conventional textual debugger.

In order to produce the animation of this algorithm with Leonardo, we embedded into the source code the following lines:

```
/**
   Graph( Out 1 );
   Node( Out N, 1) For N: InRange( N, 0, n-1 );
   Arc( U, V, 1 ) If G[ U ][ V ] != 0;
   NodePos( N, Out X, Out Y, 1 )
      Assign X = pos[ N ].x * 100
             Y = pos[ N ].y * 100;
**/
```

The goal of this visualization code is to declare a window displaying a graph with id number `1`. The vertices of the visualized graph are labeled with integers in the range $[0, n-1]$, and there is an edge `(U,V)` if and only if the corresponding entry in the adjacency matrix is non-zero. The coordinates `(x,y)` of vertex `N` of graph `1` are proportional to `pos[N].x` and `pos[N].y` respectively. The animation hinges upon the fact that, when the original algorithmic code is executed, any change in the variables `n`, `G`, and `pos` is automatically reflected in the displayed images.

Figure 2.3 illustrates different snapshots of the animation throughout the execution. Together with the window displaying the graph, there is another window showing the potential energy of each vertex (the visualization code for this window is not reported). As can easily be seen from the right column in Figure 2.3, the implementation seems to be looping among different energy configurations while trying to position vertex 0: in particular, the animation shows that vertex 0 is oscillating between two different positions. This was more difficult to discover without visualizing the running code, since the relevant values of `pos[ 0 ].x` and `pos[ 0 ].y` were never identical in the sequence of cycling steps. We also found examples where the oscillation was much more complicated, i.e., it involved more than one vertex and its periodicity was ranging over dozens of iterations. A simple analysis of the implementation code pointed out that the oscillating behavior was caused by

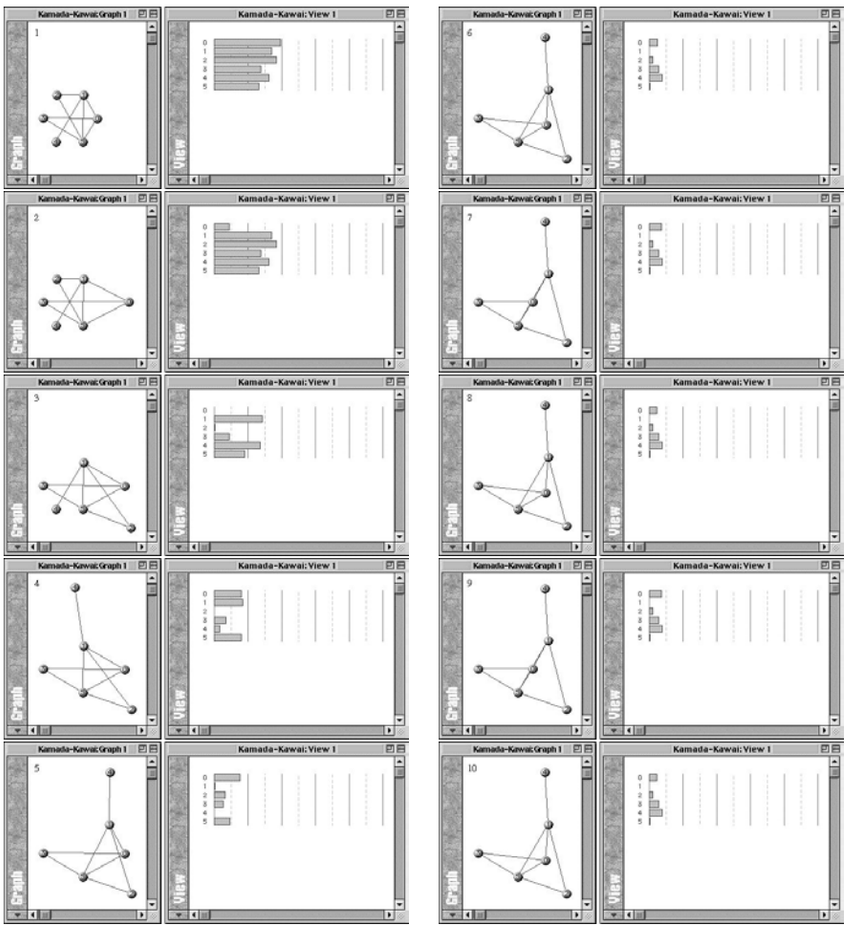


Fig. 2.3. Storyboard of Kamada and Kawai's algorithm animated with Leonardo

numerical errors: a more careful tuning of the convergence parameters was able to fix the problem.

2.4.3 Animation Systems and Demos: Geometric Algorithms

The visual nature of geometric applications makes them a natural area for designing systems that describe relevant aspects of the algorithm behavior by using animation. Indeed, the animation of geometric algorithms is mentioned among the strategic research directions in computational geometry [2.47] and increasing attention has been put towards designing algorithm visualization tools for computational geometry (see, e.g., [2.2, 2.4, 2.27, 2.46]).

In this section we show how to use the GeoWin data type introduced in Section 2.2, which was designed to be easily interfaced with algorithmic software libraries such as CGAL [2.22] and LEDA [2.34]. In particular, we discuss two of the basic features of GeoWin.

Result Scenes. A *result scene* is a GeoWin scene that depends on one or more *input scenes*. The dependence is defined by a function to be called for the objects of the input scenes. The contents of the result scene are just the output of this function. Whenever the input scene is modified the output scene is recomputed. In this way, it is very easy to write programs for showing the result of an algorithm on-line while the user is modifying the input of the algorithm, for example, by moving objects around, or by inserting or deleting objects of the input scenes.

The following piece of code shows an example program using this approach. We assume that there is a function `INTERSECT` computing the intersection points (of some type `point_t`) of a given set of straight line segments (of some type `segment_t`). Then we can create a the result scene that depends on an input scene `sc_input` of points by calling `gw.new_scene(INTERSECT, sc_input)`. Many demonstration programs in LEDA and CGAL are written in this way. In particular, all algorithms working on an input set of points (e.g., all kinds of Voronoi and Delaunay diagrams) can be visualized in a single elegant program.

```
void INTERSECT(const list<segment_t>&, list<point_t>&);

int main() {
  GeoWin gw("Segment Intersection");
  list<segment_t> L;
  geo_scene sc_input = gw.new_scene(L);
  geo_scene sc_ouput = gw.new_scene(INTERSECT,sc_input);
  gw.set_color(sc_output,red );
  gw.set_visible(sc_ouput,true );
  gw.edit(sc_input);
  return 0;
}
```

Event Handling. Every edit operation of the interactive interface of GeoWin has an associated *event*. For instance, creating a new object triggers a *new_object* event, deleting an object causes a *del_object* event, and moving an object around creates a *move_object* event. Application programs can handle these events by specifying corresponding call-back functions that are to be called whenever a certain event occurs. We show how to use event handling in the animation of a sweep line algorithm.

The program creates a special scene `sc_sweep` that contains a single vertical line, the sweep line, and it associates a call-back function `sweep_handler` with the `move_object` events of this scene (by calling `gw.set_move_handler (sc_sweep,sweep_handler)`). Now, during the interactive mode, the user can grab and move the sweep line with the mouse, and for each motion event the

sweep handler function is called, with the relative distance vector of the motion. Note that the call-back function associated with move object events has a boolean return type. The result of this function is evaluated by GeoWin and controls whether the actual motion is really executed. In the sweep example we use this fact to prevent any backward motion of the sweep line.

```
void sweep_handler(GeoWin& gw, const line& sl,
                   double dx, double dy) {
  // move sweep line horizontally by dx"
  // do not allow backward motions
  if (dx > 0) {
    sweep_x += dx;
    "process all events left of sweep_x"
  }
}

int main() {
  GeoWin gw("Sweep Demo");

  list<line> sweep_line;
  sweep_line.append(line(point(0,-100), point(0,100)));

  geo_scene sc_sweep =  gw.new_scene(sweep_line);
  gw.set_move_handler(sc_sweep, sweep_handler);
  gw.edit(sc_sweep);

  return 0;
}
```

The screenshot of Figure 2.4 shows the window of an animation that uses this technique for the animation of Fortune's sweep algorithm (see [2.23]) for computing the Voronoi Diagram of a set of points in the plane. This animation allows the user to drag the sweep line across the plane while watching several different structures: the constructed Delaunay triangulation, the shore line of parabolic arcs, and the circle events of the sweep.

2.4.4 Animation Systems and Fast Prototyping

Many animation systems require heavy modifications to the source code at hand and, in some instances, even require writing the entire animation code in order to produce a desired algorithm visualization. Thus, a user of these systems is supposed to invest a considerable amount of time writing code for the animation but also needs to have a significant algorithmic background to understand the details of the program to be visualized. This is not desirable, especially when algorithm animation is to be used in program development and debugging. Indeed, our own experience supports the same conclusions drawn in reference [2.37], namely that the effort required to animate an algorithm is one of the main factors limiting the diffusion of animation techniques

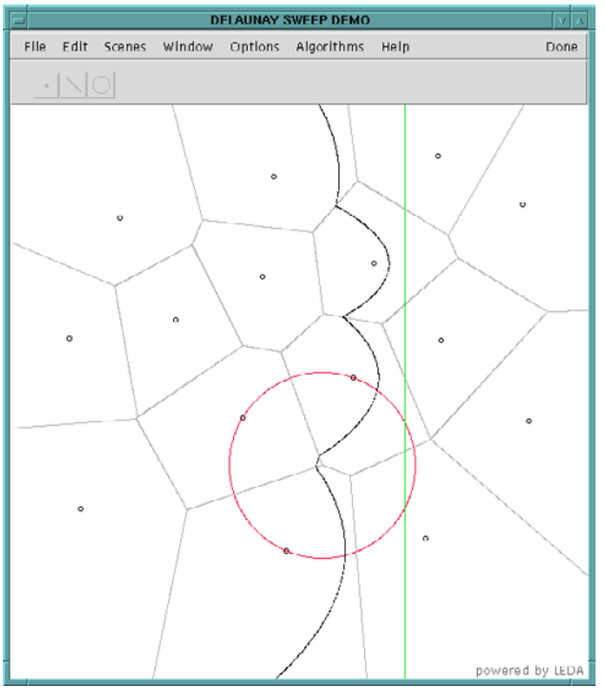


Fig. 2.4. Animation of Fortune's sweep algorithm with GeoWin

as a successful tool for debugging, testing and understanding computer algorithms.

In this section we address the issue of fast prototyping in algorithm animation and we show how this can be achieved by a deep use of *reusability*: in many cases, in the area of algorithm animation reusability is not considered at all, and very often the animation is so heavily embedded in the algorithm itself that not much of it can be reused in other animations. To achieve this, we need to enforce reusability in a strong sense: if the user produces a given *animated data type* (e.g., a stack, a tree, or a graph), then all its instances in any context (local scope, global scope, different programs) must show some standard graphical behavior with no additional effort at all. Of course, when multiple instances of different data types are animated for different goals, a basic graphical result may be poor without an additional, application-specific coordination effort that by its own nature seems not (and perhaps could never be) reusable. A successful approach is to offer different levels of sophistication: non-sophisticated animations should be basically obtained for free. If one wants a more sophisticated animation, for instance by exploiting some

coordination among different data types for the algorithm at hand, then some additional effort should be required.

We now exemplify how fast prototyping and reusability can be addressed in an animation system by taking the example of CATAI [2.14]. In particular, we describe the general steps that must be followed for preparing an animation in CATAI and at the same time illustrate them through a working example: the animation of Kruskal's algorithm for computing a minimum spanning tree (MST) of a graph [2.30].

Kruskal's algorithm first sorts all the edges by increasing cost, and then grows a forest that will finally converge to a minimum spanning tree. Initially, the forest consists of n singleton nodes (i.e., the vertices in the graph). At each step, the algorithm analyzes one edge at the time, in increasing order of their cost. If the endpoints of the edge under examination are already connected in the current forest, this edge is discarded. Otherwise, the endpoints are in two different trees: the edge is inserted into the forest (i.e., it will be a minimum spanning tree edge), and the two trees are merged into one. For efficiency issues, the trees are maintained as set union data types [2.48]. We refer to LEDA's implementation of Kruskal's algorithm [2.34], which makes use of the class `partition` to implement set union data types.

While building an algorithm animation, the first decision to be taken is which data types are to be animated. In the example at hand, for instance, it seems natural to visualize the graph being explored; additionally, we could also choose to animate the underlying partition given by the set union data types. Once this has been decided, the process of developing an animation can be broken into three different steps.

Animation Libraries. A crucial module that provides the basic tools for animation in CATAI, e.g., the graphical vocabulary, is given by the animation libraries. CATAI supplies animation libraries for most textbook algorithms: these libraries are totally independent from the data structures being animated and can be easily reused. In our example of minimum spanning trees, CATAI already contains animation libraries to represent graph objects, and thus this task is trivial.

Animated Data Types. Once animation libraries are available, we need to revise the implementation of the original data types to support some animation capabilities. We call *animated classes* the classes that implement data types with support for animation: CATAI offers a specialized `C++` library to assist in the development of animated classes. The principal component of this library is the `Animator` class, which provides animation server communication primitives and binding mechanisms between a data type and the related animation library. An animated class can be derived from the original non-animated class and from the `Animator` class. These primitives map data type operators to their animated counterparts.

In our minimum spanning tree example, the non-animated algorithm uses the LEDA graph and partition data types. The LEDA graph class uses a

single object that acts as a container to hold nodes and edges. To obtain the animated class, we derive the class `animgraph` from the LEDA `graph` class and from the `Animator` class. The methods that we wish to animate are those that change the graph: adding, removing and modifying edges or vertices. Apart from these methods, we could also add some extra methods for animation purposes.

Animated Algorithm. We are now ready to show how to animate the implementation of Kruskal's algorithm at hand. Starting from the original code, we replace the standard graph and partition with their animated counterparts. Next, we add some animation-specific code to highlight the behavior of the algorithm.

Original algorithm

```
...
G = new graph();
...
list<edge> MST::KRUSKAL(graph &G){
  node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      P->union_blocks(v,w);
    }
  }
  return T;
}
```

Animated algorithm

```
...
G = new animgraph(sockd);
...
list<edge> MST::KRUSKAL(animgraph &G){
  anim_node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    color_edge(e, GREEN);
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      color_edge(e, BLUE);
      color_node(v, BLUE);
      color_node(w, BLUE);
      P->union_blocks(v,w);
    }
    else color_edge(e, RED);
  }
  return T;
}
```

For instance, we can choose to color *green* the edge that we are currently considering. If this edge will be included in the minimum spanning tree, then we will color it *blue*, and otherwise we will color it *red*. Endpoints of *blue* edges are colored *blue*, so that a forest of *blue* trees is visualized throughout the execution of the algorithm. This *blue* forest will converge to a minimum spanning tree. The resulting algorithm is proposed as a method of a container object, i.e., an MST class, and the public interface of this object will report the services (methods) that can be requested by the end-user. One snapshot of the animation is contained in Figure 2.5.

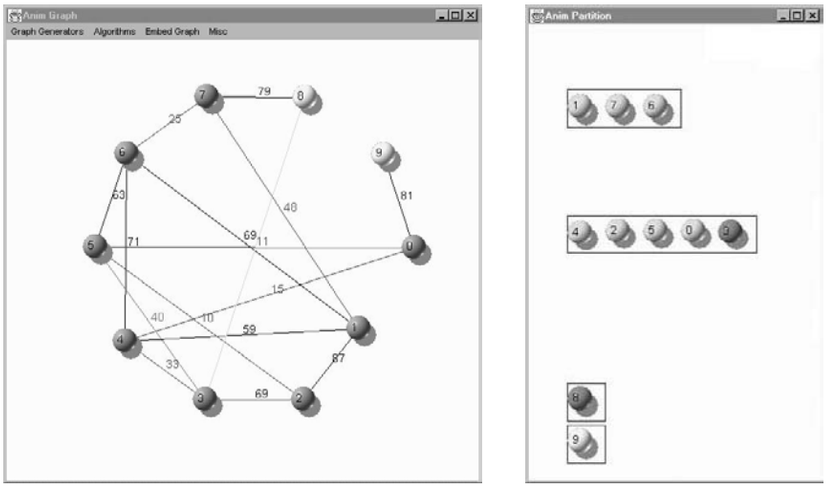


Fig. 2.5. Snapshot of the animation of Kruskal with CATAI: edges *(2,5)*, *(0,5)*, *(4,0)*, *(6,7)*, *(4,3)* and *(1,7)* have been examined and colored blue together with their endpoints, edge *(5,3)* has been colored red, and the edge *(3,8)* is currently being examined and colored green. The state of the partition is shown to the right: we have grown two blue trees (one containing vertices *0,2,3,4,5* and the other containing vertices *1,6,7*). Vertices *8* and *9* are still in singleton trees

2.5 Conclusions and Further Directions

In this paper we have addressed the role of visualization in algorithm engineering, and we have surveyed the main approaches and existing tools. Furthermore, we have discussed difficulties and relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performance of algorithmic codes.

We believe that this can have a high impact in the way we design, debug, engineer, and teach algorithms. Yet, it seems that its potential has not been fully delivered. Citing verbatim from the foreword of [2.44] by Jim Foley: "My only disappointment with the field is that software visualization has not yet had a major impact on the way we teach algorithms and programming or the way in which we debug our programs and systems. While I continue to believe in the promise and potential of software visualization, it is at the same time the case that software visualization has not yet had the impact that many have predicted and hoped for."

There are many challenges that the area of algorithm animation is currently facing. First of all, the real power of an algorithm animation system should be in the hands of the final user, possibly inexperienced, rather than of a professional programmer or of the developer of the tool. For instance, instructors may greatly benefit from fast and easy methods for tailoring ani-

mations to their specific educational needs, while they might be discouraged from using systems that are difficult to install or heavily dependent on particular software/hardware platforms. In addition to being easy to use, a software visualization tool should be able to animate significantly complex algorithmic codes without requiring a lot of effort. This seems particularly important for future development of visual debuggers. Finally, visualizing the execution of algorithms on large data sets seems worthy of further investigation. Currently, even systems designed for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck, such as changes of resolution and scale, selectivity, and elision of information.

References

2.1 R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

2.2 A. Amenta, T. Munzner, S. Levy, and M. Philips. Geomview: a system for geometric visualization. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SoCG'95)*, pages C12–C13, 1995.

2.3 R. Baecker. Sorting out sorting. In *SIGGRAPH Video Review*. Morgan Kaufmann Publishers, 1983. 30 minutes color sound film.

2.4 J. E. Baker, I. Cruz, G. Liotta, and R. Tamassia. Animating geometric algorithms over the web. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry (SoCG'96)*, pages C3–C4, 1996.

2.5 R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. Stibel. Testers and visualizers for teaching data structures. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.

2.6 A. Begeulin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing'91*, pages 435–444, 1991.

2.7 C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks*. Wiley, 1962.

2.8 M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.

2.9 M. H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, 1988.

2.10 M. H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI'88 Conference on Human Factors in Computing Systems*, pages 33–38, 1988.

2.11 M. H. Brown. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of the 7th IEEE Workshop on Visual Languages*, pages 4–9, 1991.

2.12 M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, 1992.

2.13 M. H. Brown and M. Najork. Algorithm animation using 3D interactive graphics. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 93–100, 1993.

2.14 G. Cattaneo, U. Ferraro, G. F. Italiano, and V. Scarano. Cooperative algorithm and data types animation over the net. In *Proceedings of the XV IFIP World Computer Congress*, Invited Lecture, pages 63–80, 1998. To appear in *Journal of Visual Languages and Computing*. System home page: `http://isis.dia.unisa.it/catai/`.

2.15 B. V. Cherkassky. A fast algorithm for computing maximum flow in a network. In A. V. Karzanov, editor, *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems*, pages 90-96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in *AMS Translations*, Vol. 158, pages 23–30, 1994.

2.16 B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

2.17 P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. Leonardo is available at `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

2.18 G. B. Dantzig. Application of the Simplex method to a transportation problem. In T. C. Hoopmans, editor, *Activity Analysis and Production and Allocation*, Wiley, New York, 1951.

2.19 C. Demetrescu and I. Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 12(3):253–281, 2001. The special issue devoted to selected papers from the *15th IEEE Symposium on Visual Languages (VL'99)*.

2.20 C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing algorithms over the web with the publication-driven approach. In *Proceedings of the 4th Workshop on Algorithm Engineering (WAE'00)*. Springer Lecture Notes in Computer Science 1982, 2000.

2.21 M. Eisenstadt and M. Brayshaw. The transparent Prolog Machine: an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):1–66, 1988.

2.22 A. Fabri, G. J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: a basis for geometric computation. In *Proceedings of Applied Computational Geometry: Towards Geometric Engineering Proceedings (WACG'96)*, pages 191–202, 1996.

2.23 S. J. Fortune. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the 2nd ACM Symposium on Computational Geometry (SoCG'86)*, pages 313–322, 1986.

2.24 J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proceedings of the 13th IEEE International Symposium on Visual Languages (VL'97)*, pages 360–367, 1997.

2.25 M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):23–39, 1991.

2.26 R. R. Henry, K. M. Whaley, and B. Forstall. The University of Washington Program Illustrator. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

2.27 C. A. Hipke and S. Schuierer. VEGA: a user centered approach to the distributed visualization of geometric algorithms. In *Proceedings of the 7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 110–117, 1999.

2.28 T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

2.29 K. Knowlton. *Bell Telephone Laboratories Low-Level Linked List Language*. 16-minute black and white film, Murray Hill, N.J., 1966.

2.30 J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the Amererican Mathematical Society*, 7:48–50, 1956.

2.31 S. P. Lahtinen, E. Sutinen, and J. Tarhio. Automated animation of algorithms with Eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.
2.32 H. Lieberman and C. Fry. ZStep95: a reversible, animated source code stepper. In [2.44], pages 277–292.
2.33 A. Malony and D. Reed. Visualizing parallel computer system performance. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 59–90. ACM Press, New York, 1989.
2.34 K. Mehlhorn and S. Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambrige University Press, ISBN 0-521-56329-1, 1999.
2.35 D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 1996.
2.36 B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
2.37 B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
2.38 G. C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, 1989.
2.39 G. C. Roman and K. C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.
2.40 G. C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. PAVANE: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
2.41 J. T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.
2.42 J. T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.
2.43 J. T. Stasko. Animating algorithms with X-TANGO. *SIGACT News*, 23(2):67–71, 1992.
2.44 J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.
2.45 J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.
2.46 A. Tal and D. Dobkin. Visualization of geometric algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):194–204, 1995.
2.47 R. Tamassia, P. K. Agarwal, N. Amato, D. Z. Chen, D. Dobkin, S. Drysdale, S. Fortune, M. T. Goodrich, J. Hershberger, J. O'Rourke, F. P. Preparata, J.-R. Sack, S. Suri, I. Tollis, J. S. Vitter, and S. Whitesides. Strategic directions in computational geometry. *ACM Computing Surveys*, 28(4):591–606, 1996.
2.48 R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.
2.49 B. Topol, J. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 19–26, 1995.
2.50 Q. Zhao and J. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95/01, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, 1995.

3. Parameterized Complexity: The Main Ideas and Connections to Practical Computing

Michael R. Fellows

School of Electrical Engineering and Computer Science
University of Newcastle, University Drive, Callaghan NSW 2308, Australia
mfellows@cs.newcastle.edu.au

Summary.

The purposes of this paper are two:
(1) to give an exposition of the main ideas of parameterized complexity, and
(2) to discuss the connections of parameterized complexity to the systematic design of heuristics and approximation algorithms.

3.1 Introduction

Research in the parameterized framework of complexity analysis, and on the corresponding toolkit of algorithm design methods has been expanding rapidly in recent years. This has led to a flurry of recent surveys, all of which are good sources of introductory material [3.46, 3.42, 3.22, 3.24, 3.3, 3.32, 3.33]. One could also turn to the monograph [3.21]. Experience with implementations of *FPT* algorithms is described in [3.34, 3.49, 3.3]. In several cases, these implementations now provide the best available "heuristic" algorithms for general well-known *NP*-hard problems. More importantly, the theory seems to provide some useful mathematical systematization of much existing practice in heuristics and practical computing. Computing practitioners have naturally exploited limited structural parameter ranges of the problem inputs they have been faced with, or limited parameter ranges of the solutions they have sought.

The first part of this survey summarizes the main ideas of parameterized complexity and presents a broad perspective on the program. The second part is concerned with connections to heuristics and practical computing strategies, and to approximation algorithms. Thus Section 3.2 gives the overview and main definitions. Section 3.3 explores the natural relationship of fixed-parameter tractability to the design of practical algorithms and gives several examples showing the apparently widespread situation that many industrial strength heuristics currently in use are in fact *FPT* algorithms for natural parameters, previously uncharted as such. Section 3.4 describes how parameterization according to the goodness of approximation ($k = 1/\epsilon$ for approximations to within a factor of $1 + \epsilon$ of optimal) provides a vital critical tool for evaluating the practical significance of recent work on polynomial-time approximation schemes. Section 3.5 explores how *FPT* algorithm design is

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 51–77, 2002.
© Springer-Verlag Berlin Heidelberg 2002

naturally intertwined with polynomial-time approximation algorithms and pre-processing based heuristics.

3.2 Parameterized Complexity in a Nutshell

The main ideas of parameterized complexity are organized here into two discussions:
- The basic empirical motivation.
- The perspective provided by forms of the Halting Problem.

3.2.1 Empirical Motivation: Two Forms of Fixed-Parameter Complexity

Most natural computational problems are defined on input consisting of various information, for example, many graph problems are defined as having input consisting of a graph $G = (V, E)$ and a positive integer k. Consider the following well-known problems:

Vertex Cover
Input: A graph $G = (V, E)$ and a positive integer k.
Question: Does G have a vertex cover of size at most k? (A *vertex cover* is a set of vertices $V' \subseteq V$ such that for every edge $uv \in E$, $u \in V'$ or $v \in V'$ (or both).)

Dominating Set
Input: A graph $G = (V, E)$ and a positive integer k.
Question: Does G have a dominating set of size at most k? (A *dominating set* is a set of vertices $V' \subseteq V$ such that $\forall u \in V$: $u \in N[v]$ for some $v \in V'$.)

Although both problems are *NP*-complete, the input *parameter* k contributes to the complexity of these two problems in two qualitatively different ways.

1. After many rounds of improvement involving a variety of ideas, starting from a simple $O(2^k n)$ algorithm, the best known algorithm for Vertex Cover now runs in time $O(1.271^k + kn)$ [3.17]. This is implemented and practical for n of unlimited size and k up to around 400 [3.34, 3.49, 3.19].
2. The best known algorithm for Dominating Set is *still* just the brute force algorithm of trying all k-subsets. For a graph on n vertices this approach has a running time of $O(n^{k+1})$.

Table 3.2.1 shows the contrast between these two kinds of complexity.

In order to formalize the difference between Vertex Cover and Dominating Set we make the following basic definitions.

Definition 3.2.1. *A parameterized language L is a subset $L \subseteq \Sigma^* \times \Sigma^*$. If L is a parameterized language and $(x, k) \in L$ then we will refer to x as the* main part*, and refer to k as the* parameter*.*

Table 3.1. The ratio $\frac{n^{k+1}}{2^k n}$ for various values of n and k

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	625	2,500	5,625
$k = 3$	15,625	125,000	421,875
$k = 5$	390,625	6,250,000	31,640,625
$k = 10$	1.9×10^{12}	9.8×10^{14}	3.7×10^{16}
$k = 20$	1.8×10^{26}	9.5×10^{31}	2.1×10^{35}

A parameter may be non-numerical, and it can also represent an aggregate of various parts or structural properties of the input.

Definition 3.2.2. *A parameterized language L is* multiplicatively fixed-parameter tractable *if it can be determined in time $f(k)q(n)$ whether $(x, k) \in L$, where $|x| = n$, $q(n)$ is a polynomial in n, and f is a function (unrestricted).*

Definition 3.2.3. *A parameterized language L is* additively fixed-parameter tractable *if it can be determined in time $f(k)+q(n)$ whether $(x, k) \in L$, where $|x| = n$, $q(n)$ is a polynomial in n, and f is a function (unrestricted).*

As an exercise, the reader might wish to show that a parameterized language is additively fixed-parameter tractable if and only if it is multiplicatively fixed-parameter tractable. This emphasizes how cleanly fixed-parameter tractability isolates the computational difficulty in the complexity contribution of the parameter.

There are many ways that parameters arise naturally, for example:

• *The size of a database query.* Normally the size of the database is huge, but frequently queries are small. If n is the size of a relational database, and k is the size of the query, then answering the query (Model Checking) can be solved trivially in time $O(n^k)$. It is known that this problem is unlikely to be *FPT* [3.23, 3.44] because it is hard for $W[1]$ (a form of negative evidence explained in Section 3.2.2). However, if the parameter is the size of the query and the treewidth of the database, then the problem is fixed-parameter tractable. It appears that many databases encountered in practice do have bounded treewidth, so this quite nontrivial *FPT* result has significant potential to be useful [3.33].

• *The nesting depth of a logical expression.* ML compilers work reasonably well. One of the problems the compiler must solve is the checking of the compatibility of type declarations. This problem is complete for deterministic exponential time [3.35], so the situation appears dire from the standpoint of complexity theory. The implementations work well in practice, using an algorithm that previously would have been called a heuristic because — we can now say — the ML Type Checking problem is solved by an *FPT* algorithm with a running time of $O(2^k n)$, where n is the size of the program

and k is the maximum nesting depth of the type declarations [3.39]. Since normally $k \leq 6$, the algorithm is clearly practical.

• *The number of species in an evolutionary tree.* Frequently this parameter is in a range of $k \leq 50$. The PHYLIP computational biology server includes an algorithm which solves the STEINER PROBLEM IN HYPERCUBES in order to compute possible evolutionary trees based on (binary) character information. The exponential heuristic algorithm that is used is in fact an *FPT* algorithm when the parameter is the number of species [3.24].

• *The number of processors in a practical parallel processing system.* This is frequently in the range of $k \leq 64$. Is there a practical and interesting theory of parallel *FPT*? For a recent paper that explores practical parallel implementations of *FPT* algorithms see [3.19]. (Coarse-grained parallel implementations of *FPT* algorithms for VERTEX COVER are accessible on an algorithmic server at the University of Carleton website.)

• *The number of variables or clauses in a logical formula, or the number of steps in a deductive procedure.* Some initial studies of applications of parameterized complexity to logic programming and artificial intelligence have recently appeared [3.50, 3.30]. Much remains unexplored. Determining whether at least k clauses of a CNF formula F are satisfiable is *FPT* with a running time of $O(|F| + 1.381^k k^2)$ [3.8]. Since at least half of the m clauses of F can always be satisfied, a more natural parameterization is to ask if at least $m/2 + k$ clauses can be satisfied — this is *FPT* with a running time of $O(|F| + 6.92^k k^2)$ [3.8]. Implementations indicate that these algorithms are quite practical [3.31], with the kernelization transformations of the *FPT* algorithms having particular practical value in decreasing the size of the subsequent exponential search trees.

• *The number of steps for a motion planning problem.* Where the description of the terrain has size n (which therefore bounds the number of movement options at each step), we can solve this problem in time $O(n^{k+1})$ trivially. Are there significant classes of motion planning problems that are fixed-parameter tractable? Exploration of this topic has hardly begun [3.14].

• *The number of moves in a game, or the number of steps in a planning problem.* While most game problems are *PSPACE*-complete classically, it is known that some are *FPT* and others are likely not to be *FPT* (because they are hard for $W[1]$), when parameterized by the number of moves of a winning strategy [3.1]. The size n of the input game description usually governs the number of possible moves at any step, so there is a trivial $O(n^k)$ algorithm that just examines the k-step game trees exhaustively. This is potentially a very fruitful area, since games are used to model many different kinds of situations.

• *The number of facilities to be located.* Determining whether a planar graph has a dominating set of size at most k is fixed-parameter tractable by an algorithm with a running time of $O(8^k n)$ based on kernelization and search

trees. By different methods, an *FPT* running time of $O(3^{36\sqrt{k}})n$ can also be proved. This does not appear to be practical on the basis of this parameter function $f(k)$, but the algorithm has been implemented and appears to be practical for k up to 500 for classes of randomly generated tests. This seems to be the best available algorithm for Planar Dominating Set.

• *A "dual" parameter.* A graph has an independent set of size k if and only if it has a vertex cover of size $n-k$. Many problems have such a natural dual form and it is "almost" a general rule, first noted by Raman, that parametric duals of *NP*-hard problems have complementary parameterized complexity (one is *FPT*, and the other is $W[1]$-hard) [3.38, 3.6]. For example, $n-k$ Dominating Set is *FPT*, as is $n-k$ Graph Coloring. Solving a hard problem by parameterizing from "the other end" appears to be an important and general algorithmic strategy. The best available algorithm for Maximum Independent Set is to compute a Vertex Cover by the very good *FPT* algorithms for this problem, and take the complement.

• *An unrelated parameter.* The input to a problem might come with "extra information" because of the way the input arises. For example, we might be presented with an input graph G together with a k-vertex dominating set in G, and be required to compute an optimal bandwidth layout. Whether this problem is *FPT* is open. Problems of this sort have recently begun to receive attention [3.11].

• *The amount of "dirt" in the input or output for a problem.* In the Maximum Agreement Subtree (MAST) problem we are presented with a collection of evolutionary trees trees for a set X of species. These might be obtained by studying different gene families, for example. Because of errors in the data, the trees might not be isomorphic, and the problem is to compute the largest possible subtree on which they do agree. Parameterized by the number of species that need to be deleted to achieve agreement, the MAST problem is *FPT* by an algorithm having a running time of $O(2.27^k + rn^3)$ where r is the number of trees and n is the number of species [3.43].

• *The "robustness" of a solution to a problem, or the distance to a solution.* For example, given a solution of the Minimum Spanning Tree problem in an edge-weighted graph, we can ask if the cost of the solution is robust under all increases in the edge costs, where the parameter is the total amount of cost increases.

• *The distance to an improved solution.* Local search is a mainstay of heuristic algorithm design. The basic idea is that one maintains a *current solution*, and iterates the process of moving to a neighboring "better" solution. A neighboring solution is usually defined as one that is a single step away according to some small edit operation between solutions. The following problem is completely general for these situations, and could potentially provide a valuable subroutine for "speeding up" local search:

k-Speed Up for Local Search
Input: A solution S, k.
Parameter: k
Output: The best solution S' that is within k edit operations of S.

Is it *FPT* to explore the k-change neighborhood for TSP?

• *The goodness of an approximation.* If we consider the problem of producing solutions whose value is within a factor of $(1 + \epsilon)$ of optimal, then we are immediately confronted with a natural parameter $k = 1/\epsilon$. Many of the recent PTAS results for various problems have running times with $1/\epsilon$ in the exponent of the polynomial. Since polynomial exponents larger than 3 are not practical, this is a crucial parameter to consider. The reader will find more about this in Section 3.4.

It is obvious that the practical world is full of concrete problems governed by parameters of all kinds that are bounded in small or moderate ranges. If we can design algorithms with running times like $2^k n$ for these problems, then we may have something really useful.

The following definition provides us with a place to put all those problems that are "solvable in polynomial time for fixed k" without making our central distinction about whether this "fixed k" is ending up in the exponent or not.

Definition 3.2.4. *A parameterized language L belongs to the class XP (slicewise P) if it can be determined in time $f(k)n^{g(k)}$ whether $(x, k) \in L$, where $|x| = n$, with f and g being unrestricted functions.*

Is it possible that $FPT = XP$? This is one of the few structural questions concerning parameterized complexity that currently has an answer [3.21].

Theorem 3.2.1. *FPT is a proper subset of XP.*

3.2.2 The Halting Problem: A Central Reference Point

The main investigations of computability and efficient computability are tied to three basic forms of the Halting Problem.

1. The Halting Problem
   Input: A Turing machine M.
   Question: If M is started on an empty input tape, will it ever halt?
2. The Polynomial-Time Halting Problem for Nondeterministic Turing Machines
   Input: A nondeterministic Turing machine M.
   Question: Is it possible for M to reach a halting state in n steps, where n is the length of the description of M?
3. The k-Step Halting Problem for Nondeterministic Turing Machines
   Input: A nondeterministic Turing machine M and a positive integer k.

(The number of transitions that might be made at any step of the computation is unbounded, and the alphabet size is also unrestricted.)
Parameter: k
Question: Is it possible for M to reach a halting state in at most k steps?

The first form of the HALTING PROBLEM is useful for studying the question:

"Is there ANY algorithm for my problem?"

The second form of the HALTING PROBLEM has proved useful for nearly 30 years in addressing the question:

"Is there an algorithm for my problem ... like the ones for SORTING and MATRIX MULTIPLICATION?"

The second form of the HALTING PROBLEM is trivially *NP*-complete, and essentially defines the complexity class *NP*. For a concrete example of why it is trivially *NP*-complete, consider the 3-COLORING problem for graphs, and notice how easily it reduces to the *P*-TIME NDTM HALTING PROBLEM. Given a graph G for which 3-colorability is to be determined, we just create the following nondeterministic algorithm:

Phase 1. (There are n lines of code here if G has n vertices.)
(1.1) Color vertex 1 one of the three colors nondeterministically.
(1.2) Color vertex 2 one of the three colors nondeterministically.
...
(1.n) Color vertex n one of the three colors nondeterministically.

Phase 2. Check to see if the coloring is proper and if so halt. Otherwise go into an infinite loop.

It is easy to see that the above nondeterministic algorithm has the possibility of halting in m steps (for a suitably padded Turing machine description of size m) if and only if the graph G admits a 3-coloring. Reducing any other problem $\Pi \in NP$ to the *P*-TIME NDTM HALTING PROBLEM is no more difficult than taking an argument that the problem Π belongs to NP and modifying it slightly to be a reduction to this form of the HALTING PROBLEM. It is in this sense that the *P*-TIME NDTM HALTING PROBLEM is essentially the *defining* problem for NP.

The conjecture that $P \neq NP$ is intuitively well-founded. The second form of the HALTING PROBLEM would seem to require exponential time because there is little we can do to analyze unstructured nondeterminism other than to exhaustively explore the possible computation paths. Apart from accumulated habit, this concrete intuition is the fundamental reference point for classical complexity theory.

When the question is:

"Is there an algorithm for my problem ... like the one for VERTEX COVER?"

the third form of the HALTING PROBLEM anchors the discussion. This question will increasingly and inevitably be asked for any *NP*-hard problem for which small parameter ranges of input or output aspects or structure are important in applications. It is reasonable to assert that there are few applications of computing where this will not be true.

The third natural form of the HALTING PROBLEM is trivially solvable in time $O(n^k)$ by exploring the n-branching, depth-k tree of possible computation paths exhaustively. Our intuition here is essentially the same as for the second form of the Halting Problem — that this cannot be improved. The third form of the Halting Problem defines the parameterized complexity class $W[1]$. Thus $W[1]$ is strongly analogous to NP, and the conjecture that $FPT \neq W[1]$ stands on much the same intuitive grounds as the conjecture that $P \neq NP$. The appropriate notion of problem reduction is as follows.

Definition 3.2.5. *A parametric transformation from a parameterized language L to a parameterized language L' is an algorithm that computes from input consisting of a pair (x, k), a pair (x', k') such that:*

1. *$(x, k) \in L$ if and only if $(x', k') \in L'$,*
2. *$k' = g(k)$ is a function only of k, and*
3. *the computation is accomplished in time $f(k)n^\alpha$, where $n = |x|$, α is a constant independent of both n and k, and f is an arbitrary function.*

Hardness for $W[1]$ is the working criterion that a parameterized problem is unlikely to be *FPT*. The k-CLIQUE problem is $W[1]$-complete [3.21], and often provides a convenient starting point for $W[1]$-hardness demonstrations.

3.3 Connections to Practical Computing and Heuristics

What is the working context of practical computing? A thought-provoking account of this subject has been given by Weihe [3.52].

A crucial question is: *What are the actual inputs that practical computing implementations have to deal with?*

In considering "war stories" of practical computing, such as reported by Weihe, we are quickly forced to give up the idea that real inputs (for most problems) fill up the definitional spaces of our mathematical modeling. The general rule also is that real inputs are *not random*, but rather have lots of hidden structure that may not have a familiar name or conceptualization.

Example 1: Weihe's Train Problem

Weihe describes a problem concerning the train systems of Europe [3.51]. Consider a bipartite graph $G = (V, E)$ where V is bipartitioned into two sets S (stations) and T (trains), and where an edge represents that a train t stops at a station s. The relevant graphs are huge, on the order of 10,000 vertices. The problem is to compute a minimum number of stations $S' \subseteq S$ such that

every train stops at a station in S'. It is easy to see that this is a special case of the HITTING SET problem, and is therefore *NP*-complete. Moreover, it is also $W[1]$-hard [3.21], so the straightforward application of the parameterized complexity program seems to fail as well.

However, the following two reduction rules can be applied to simplify (pre-process) the input to the problem. In describing these rules, let $N(s)$ denote the set of trains that stop at station s, and let $N(t)$ denote the set of stations at which the train t stops.

1. If $N(s) \subseteq N(s')$ then delete s.
2. If $N(t) \subseteq N(t')$ then delete t'.

Applications of these reduction rules cascade, preserving at each step enough information to obtain an optimal solution. Weihe found that, remarkably, these two simple reduction rules were strong enough to "digest" the original, huge input graph into a *problem kernel* consisting of disjoint components of size at most 50 — small enough to allow the problem to be solved optimally by brute force.

Note that in the same breath, we have here a polynomial-time constant factor approximation algorithm, getting us a solution within a factor of 50 of optimal in, say, $O(n^2)$ time, just by taking *all* the vertices in the kernel components.

Weihe's example displays a universally applicable coping strategy for hard problems: *smart pre-processing*. It would be silly *not* to undertake pre-processing for an *NP*-hard problem, even if the next phase is simulated annealing, neural nets, roaming ants, genetic, memetic or the kitchen sink. In a precise sense, this is *exactly* what fixed-parameter tractability is all about. The following is an equivalent definition of *FPT* [3.24].

Definition 3.3.1. *A parameterized language L is* kernelizable *if there is a parametric transformation of L to itself, and a function h (unrestricted) that satisfies:*

1. *the running time of the transformation of (x, k) into (x', k'), where $|x| = n$, is bounded by a polynomial $q(n, k)$ (so that in fact this is a polynomial-time transformation of L to itself, considered classically, although with the additional structure of a parametric reduction),*
2. *$k' \leq k$, and*
3. *$|x'| \leq h(k)$, where h is an arbitrary function.*

Lemma 3.3.1. *A parameterized language L is fixed-parameter tractable if and only if it is kernelizable.*

Weihe's example looks like an *FPT* kernelization, but what is the parameter? As a thought experiment, let us define $K(G)$ for a bipartite graph G to be the maximum size of a component of G when G is reduced according to the two simple reduction rules above. Then it is clear, although it might seem

artificial, that HITTING SET can be solved optimally in *FPT* time for the parameter $K(G)$. We can add this new tractable parameterization of HITTING SET to the already known fact that HITTING SET can be solved optimally in *FPT*-time for the parameter *treewidth*. (It is not hard to show that treewidth and $K(G)$ are unrelated.)

As an illustration of the power of pre-processing, the reader will easily discover a reduction rule for VERTEX COVER that eliminates all vertices of degree 1. Not so easy is to show that all vertices of degree at most 3 can be eliminated, leaving as a kernel a graph of minimum degree four. This pre-processing routine yields the best known heuristic algorithm for the general VERTEX COVER problem (i.e., no assumption that k is small), and also plays a central role in the best known *FPT* algorithm for VERTEX COVER [3.17].

We see in Weihe's train problem an example of a problem where the natural input distribution (graphs of train systems) occupies a limited parameter range, but the relevant parameter is not at all obvious. The inputs to one computational process (e.g., Weihe's train problem) are often the outputs of another process (the building and operating of train systems) that also are governed by computational and other feasibility constraints. We might reasonably adopt the view that the real world of computing involves a vast commerce in hidden structural parameters.

Weihe's algorithm is a beautiful example of an *FPT* algorithm that exploits a significant hidden structural parameter of the graphs that arise in analyzing train systems, and follows a "classic" pattern in *FPT* algorithm design: (1) a *P*-time kernelization, pre-processing phase, followed by (2) an exponential search phase (exponential in the size of the kernel).

Example 2: Heuristics for Breakpoint Phylogeny

It is assumed that there is a fixed set $\mathcal{G}$ of genes shared by all the species for which an evolutionary tree is to be constructed. With each species S is associated a circular ordering of $\mathcal{G}$ where each gene occurs signed, either positively or negatively, according to the transcription direction for the gene (and each gene occurs in the circular ordering for the species exactly once). Given two such circular orderings for species S and S', a *breakpoint* is an ordered pair of genes (g, g') such that g and g' occur consecutively in S in that order, but neither (g, g') nor $(-g', -g)$ occur consecutively in that order in S'. The *breakpoint distance* $d(S, S')$ between S and S' is the number of breakpoints between S and S'. The *breakpoint score* of a tree labeled at each node with a signed circular ordering of $\mathcal{G}$ is the sum of the breakpoint distances along the edges of the tree.

The BREAKPOINT PHYLOGENY problem takes as input a set of signed circular orderings $S_1, ..., S_n$ and seeks to find a tree T with n leaves such that:

1. The leaves are labeled 1:1 with the S_i.
2. The internal vertices are labeled with signed circular orderings of $\mathcal{G}$ that represent hypothesized ancestral species.
3. The breakpoint score of T is minimized.

The problem is apparently quite difficult; it is *NP*-complete, even for the seemingly severe restriction to $n = 3$ — only three species! — known as the Beakpoint Median Problem [3.45]. A substantial speedup is reported by Moret, et al. [3.40], for a heuristic based on the following kernelization rule.

– If two genes *always* occur consecutively (either as (g, g') or as $(-g', -g)$) in the circular ordering of each of the species under consideration, then "fuse" g and g' into a single replacement *metagene*. (Thus the size of $\mathcal{G}$, and the length of the circular orderings, is effectively decreased by one.)

The authors of [3.40] report that this single reduction rule yields a speedup by a factor of 6 over the implementation of Blanchette, Bourque and Sankoff [3.10].

This first phase of kernelization on a Campanulaceae data set of 13 species with an initial set of genes $\mathcal{G}$ of size $|\mathcal{G}| = 150$ was found to simplify the input to a set $\mathcal{G}'$ of metagenes of size $|\mathcal{G}'| = 35$. Following this initial kernelization, the second phase considers a sampling of all $(2n - 5)!!$ leaf-labelled trees on n leaves, and for each of these uses a separate heuristic to explore possible internal labellings.

This heuristic was not developed as an *FPT* algorithm, yet it is one, for a realistic natural parameter — the total cost of the tree.

Theorem 3.3.1. *The* Breakpoint Median Problem *is fixed-parameter tractable for the parameter k taken to be the total cost of the tree.*

Proof. If the tree has total cost k then it has at most k internal edges (since each contributes some cost to the total) and therefore at most $k - 1$ leaves and at most $k - 2$ internal vertices. It is also not hard to see that after kernelization, the number of genes in each sequence is at most k. Thus, the cost of exhaustively checking each of the $(2k-7)!!$ leaf-labelled trees on $k-1$ leaves, and exhaustively trying all possible assignments of the $k!$ possible gene orderings (of the kernelized instance) to the internal vertices of the trees — all of this can be accomplished in time bounded by a function of k. □

The running time of the kernelization + brute force *FPT* algorithm described above would not seem conducive to practical implementation, since the implicit parameter function $f(k)$ is around $(k!)^k$, which exceeds 10^{20} when $k = 4$. What Moret *et al.* have implemented is essentially a heuristic adaptation of this *FPT* algorithm, based on a sampling of the possible trees and a sampling of the possible internal vertex assignments.

General heuristic design strategies that correspond to some of the main *FPT* methods are displayed in Table 3.3. The essential theme is to obtain heuristic methods from *FPT* algorithms by strategies for *deflating the parametric costs* by truncating or sampling the search trees or obstruction sets, etc.

Table 3.2. Some *FPT* methods and heuristic strategies

FPT Technique	Heuristic Design Strategy
Reduction to a Problem Kernel	A useful pre-processing subroutine for any heuristic.
Search Tree	Explore only an affordable, heuristically chosen subtree.
Well-Quasiordering	Use a sample of the obstruction set.
Color-Coding	Use a sample of the hash functions.

The following problem is also fixed-parameter tractable.

The Steiner Problem for Generalized Hypercubes
Instance: The input the problem consists of the following pieces of information:

1. A set of complete weighted digraphs D_i for $i = 1, ..., n$, each described by a set of vertices V_i and a function

$$t_i : V_i \times V_i \rightarrow I\!N$$

(We refer to the vertices of D_i as *character states*, to D_i as the *character state digraph*, and to t_i as the *state transition cost function* for the ith character.)
2. A positive integer k_1 such that $|V_i| \leq k_1$ for $i = 1, ..., n$.
3. A set X of k_2 length n vectors x_j for $j = 1, ..., k_2$, where the ith component $x_j[i] \in V_i$. That is, for $j = 1, ..., k_2$,

$$x_j \in \Omega = \prod_{i=1}^{n} V_i$$

4. A positive integer M.

Parameter: (k_1, k_2)
Question: Is there a rooted tree $T = (V, E)$ and an assignment to each vertex $v \in V$ of T of an element $y_v \in \Omega$, such that:

– X is assigned one-to-one to the set of leaves of T,
– The sum over all parent-child edges uv of T, of the *total transition cost* for the edge, defined to be

$$\sum_{i=1}^{n} t_i(y_u[i], y_v[i])$$

is bounded by M?

Theorem 3.3.2. The Steiner Problem for Generalized Hypercubes *is fixed-parameter tractable.*

Proof. We define an equivalence relation $i \sim j$ on the index space $\{1, ..., n\}$ that allows us to combine D_i and D_j and obtain an equivalent smaller instance. In order to define $\sim$ we first define some other equivalences.

Fix $m \leq k_1$ and let l be an integer edge labeling of the complete digraph K_m on m vertices. Let $v_1, ..., v_m$ denote the vertices of K_m. Let T be a rooted tree with k_2 leaves labeled from $v_1, ..., v_m$. Define the *cost* of T with respect to l to be the minimum, over all possible labelings s of the internal vertices of T with labels taken from $\{v_1, ..., v_m\}$, of the sum over the parent-child edges of T of the transition costs given by l on the labels, and write this as

$$\text{cost}(T, l) = \min_s\{\text{cost}(T, s, l)\}$$

If l and l' are integer edge labelings of K_m and T is as above, then define $l \sim_T l'$ if and only if $\exists s$ such that

$$\text{cost}(T, l) = \text{cost}(T, s, l) = \text{cost}(T, s, l') = \text{cost}(T, l')$$

and define $l \sim l'$ if and only $l \sim_T l'$ for all such trees T.

For $i, i' \in \{1, ..., n\}$ define $i \sim i'$ if and only if:

1. $|V_i| = |V_{i'}| = m$ so that the only difference between D_i and $D_{i'}$ is in their arc-labelings l and l', and
2. $l \sim l'$.

The kernelization algorithm can now be described quite simply. Let $\mathcal{I}$ be an instance of the problem. If there are indices $i \neq i'$ for which $i \sim i'$, then modify $\mathcal{I}$ by combining these into one character state digraph with the state transition cost function given by the arc-labeling given $l + l'$, where these are the cost functions for D_i and $D_{i'}$, respectively. Let $\mathcal{I}'$ denote the modified instance.

The correctness of the reduction to the smaller instance is obvious. We need only to note that the equivalence $i \sim i'$ can be determined in time bounded by a function of the parameter and that number of equivalence classes is similarly bounded by a function of the parameter. □

The parameter function for this simple kernelization-based *FPT* algorithm is nearly as discouraging as the one for Breakpoint Phylogeny. We remark that most of the expense is in determining when two transition digraph indices i and i' are equivalent by testing them on all possible trees with k_2 leaves. This suggests a heuristic algorithm that combines indices when they fail to be distinguished by a (much smaller) random sample of trees and leaf-labelings.

A previous survey described this *FPT* result in the context of an encounter with an evolutionary biologist who reported earlier, rather fruitless interactions with theoretical computer scientists who proved that his problems were *NP*-complete and "went away". We claimed that we were different! and that

we had a result on one of his computational problems (The Steiner Problem for Hypercubes) that might be of interest. After we described the *FPT* algorithm he said simply [3.27]:

"That's what I already do!"

Those who design *FPT* algorithms should keep in mind that their $f(k)$'s are only the best they are able to prove concerning a worst-case analysis, and that their algorithms may in fact be much more useful in practice than the pessimistic analysis indicates, on realistic inputs, particularly if any nontrivial kernelization is involved. Furthermore, large parametric costs can also be systematically mitigated in heuristic adaptations of *FPT* algorithms. Real usefulness can only be settled by implementation and experimentation.

3.4 A Critical Tool for Evaluating Approximation Algorithms

The emphasis in the substantial new industry of research on polynomial-time approximation algorithms is concentrated on the notions of:

- Polynomial-time constant factor approximation algorithms.
- Polynomial-time approximation schemes.

The connections between the *parameterized complexity* and *polynomial-time approximation* programs are deep and developing rapidly. Approximation immediately concerns a fundamental parameter: $k = 1/\epsilon$, *the goodness of the approximation.*

To illustrate the issue, consider the following more-or-less random sample of recent PTAS results:

- The PTAS for the Euclidean TSP due to Arora [3.4] has a running time of around $O(n^{3000/\epsilon})$. Thus for a 20% error, we have a "polynomial-time" algorithm that runs in time $O(n^{15000})$.
- The PTAS for the Multiple Knapsack problem due to Chekuri and Khanna [3.16] has a running time of $O(n^{12(\log(1/\epsilon)/\epsilon^8)})$. Thus for a 20% error we have a "polynomial-time" algorithm that runs in time $O(n^{9375000})$.
- The PTAS for the Minimum Cost Routing Spanning Tree problem due to Wu, Lancia, Banfna, Chao, Ravi and Tang [3.53] has a running time of $O(n^{2\lceil 2/\epsilon \rceil - 2})$. For a 20% error, we thus have a "polynomial" running time of $O(n^{18})$.
- The PTAS for the Unbounded Batch Scheduling problem due to Deng, Feng, Zhang and Zhu [3.20] has a running time of $O(n^{5\log_{1+\epsilon}(1+(1/\epsilon))})$. Thus for a 20% error we have an $O(n^{50})$ polynomial-time algorithm.
- The PTAS for Two-Vehicle Scheduling on a Path due to Karuno and Nagamochi [3.36] has a running time of $O(n^{8(1+(2/\epsilon))})$; thus we have $O(n^{88})$ time for a 20% error.

– The PTAS for the Maximum Subforest Problem due to Shamir and Tsur [3.48] has a running time of $O(n^{2^{2^{1/\epsilon}}-1})$. For a 20% error we thus have a "polynomial" running time of $O(n^{958267391})$.
– The PTAS for the Maximum Indendent Set problem on geometric graphs due to Erlebach, Jansen and Seidel [3.25] has a running time of $O(n^{(4/\pi)(1/\epsilon^2+2)^2(1/\epsilon^2+1)^2})$. Thus for a 20% error we have a running time of $O(n^{532804})$.
– The PTAS for the Class-Constrained Packing Problem due to Shachnai and Tamir [3.47] has a running time of $O(n^{64/\epsilon+(log(1/\epsilon)/\epsilon^8)})$ (for three colors). Thus for a 20% error (for three colors) we have a running time of $O(n^{1021570})$.
– The PTAS for the problem of Base Station Positioning in UMTS Networks due to Galota, Glasser, Reith and Vollmer [3.28] has a running time of $O(n^{25/\epsilon^2})$, and thus we have $O(n^{627})$ time for a 20% error.
– The PTAS for the General Multiprocessor Job Scheduling Problem due to Chen and Miranda [3.18] runs in time $O(n^{(3mm!)^{(m/\epsilon)+1}})$ for m machines. Thus for 4 machines with a 20% error we have an algorithm that runs in time $O(n^{10000000000000000000000000000000000000000000000000000000000000000})$ or so.

Since polynomial-time algorithms with exponent greater than three are generally not very practical, the following question would seem to be important.

Can we get the $k = 1/\epsilon$ out of the exponent?

The following definition captures the essential issue.

Definition 3.4.1. *An optimization problem Π has an* efficient P-time approximation scheme *(EPTAS) if it can be approximated to a goodness of $(1 + \epsilon)$ of optimal in time $f(k)n^c$ where c is a constant and $k = 1/\epsilon$.*

In 1997, Arora gave an EPTAS for the Euclidean TSP [3.5], but for all of the other PTAS's mentioned above, the possibility of such an improvement remains open, and perhaps not much explored, particularly in terms of potential $W[1]$-hardness.

The following easy but important connection between parameterized complexity and approximation was first proved by Bazgan [3.9, 3.13].

Theorem 3.4.1. *Suppose that Π_{opt} is an optimization problem, and that Π_{param} is the corresponding parameterized problem, where the parameter is the value of an optimal solution. Then Π_{param} is fixed-parameter tractable if Π_{opt} has an efficient PTAS.*

Applying Bazgan's Theorem is not necessarily difficult — we will sketch here a recent example. Khanna and Motwani introduced three planar logic problems in an interesting effort to give a general explanation of PTAS-approximability. Their suggestion is that "hidden planar structure" in the

logic of an optimization problem is what allows PTASs to be developed [3.37]. They gave examples of optimization problems known to have PTASs, problems having nothing to do with graphs, that could nevertheless be reduced to these planar logic problems. The PTASs for the planar logic problems thus "explain" the PTASs for these other problems. Here is one of their three general planar logic optimization problems.

PLANAR TMIN
Input: A collection of Boolean formulas in sum-of-products form, with all literals positive, where the associated bipartite graph is planar (this graph has a vertex for each formula and a vertex for each variable, and an edge between two such vertices if the variable occurs in the formula).
Output: A truth assignment of minimum weight (i.e., a minimum number of variables set to *true*) that satisfies all the formulas.

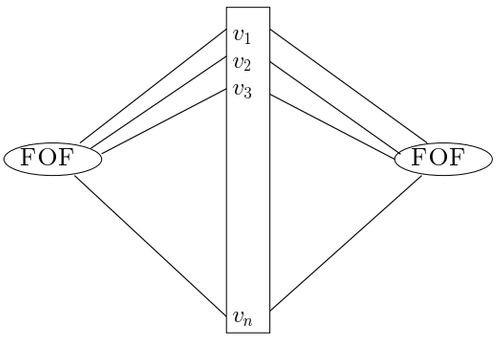
The following theorem is recent joint work with Cai, Juedes and Rosamond [3.12].

Theorem 3.4.2. *Planar TMIN is hard for* $W[1]$ *and therefore does not have an EPTAS unless* $FPT = W[1]$.

Proof. We show that CLIQUE is parameterized reducible to PLANAR TMIN with the parameter being the weight of a truth assignment. Since CLIQUE is W[1]-complete, it will follow that the parameterized form of PLANAR TMIN is W[1]-hard.

To begin, let $\langle G, k\rangle$ be an instance of CLIQUE. Assume that G has n vertices. From G and k, we will construct a collection C of FOFs (sum-of-products formulas) over $f(k)$ blocks of n variables. C will contain at most $2f(k)$ FOFs and the incidence graph of C will be planar. Moreover, each minterm in each FOF will contain at most 4 variables. The collection C is constructed so that G has a clique of size k if and only if C has a weight $f(k)$ satisfying assignment with exactly one variable set to true in each block of n variables. Here we have that $f(k) = O(k^4)$.

To maintain planarity in the incidence graph for C, we ensure that each block of n variables appears in at most 2 FOFs. If this condition is maintained, then we can draw each block of n variables as follows.

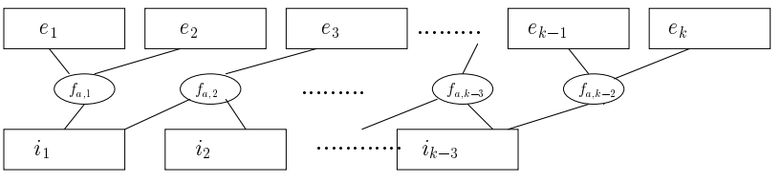


We describe the construction in two stages. In the first stage, we use k blocks of n variables and a collection C' of $k(k-1)/2+k$ FOFs. In a weight k satisfying assignment for C', exactly one variable v_i, j in each block of variables $b_i = [v_{i,1}, \ldots, v_{i,n}]$ will be set to true. We interpret this event as "vertex j is the ith vertex in the clique of size k." The $k(k-1)/2+k$ FOFs are described as follows. For each $1 \leq i \leq k$, let f_i be the FOF $\bigvee_{j=1}^{n} v_{i,j}$. This FOF ensures that at least one variable in b_i is set to true. For each pair $1 \leq i < j \leq k$, let $f_{i,j}$ be the FOF $\bigvee_{(u,v)\in E} v_{i,u}v_{j,v}$. Each FOF $f_{i,j}$ ensures that there is an edge in G between the ith vertex the clique and the jth vertex in the clique.

It is straightforward to show that $C' = \{f_1, \ldots, f_k, f_{1,2}, \ldots, f_{k-1,k}\}$ has a weight k satisfying assignment if and only if G has a clique of size k. To see this, notice that any weight k satisfying assignment for C' must satisfy exactly one variable in each block b_i. Each first order formula $f_{i,j}$ ensures that there is an edge between the ith vertex in the potential clique and the jth vertex in the potential clique. Notice also that, since we assume that G does not contain edges of the form (u, u), the FOF $f_{i,j}$ also ensures that the ith vertex in the potential clique is not the jth vertex in the potential clique. This completes the first stage.

The incidence graph for the collection C' in the first stage is almost certainly not planar. In the second stage, we achieve planarity by removing crossovers in incidence graph for C'. Here we use two types of widgets to remove crossovers while keeping the number of variables per minterm bounded by four. The first widget A_k consists of $k + k - 3$ blocks of n variables and $k - 2$ FOFs. This widget consists of $k - 3$ internal and k external blocks of variables. Each external block $e_i = [e_{i,1}, \ldots, e_{i,n}]$ of variables is connected to exactly one FOF inside the widget. Each internal block $i_j = [i_{j,1}, \ldots, e_{j,n}]$ is connected to exactly two FOFs inside the widget. The $k - 2$ FOFs are given as follows. The FOF $f_{a,1}$ is $\bigvee_{j=1}^{n} e_{1,j}e_{2,j}i_{1,j}$. For each $2 \leq l \leq k-3$, the FOF $f_{a,l} = \bigvee_{j=1}^{n} i_{l-1,j}e_{l+1,j}i_{l,j}$. Finally, $f_{a,k-2} = \bigvee_{j=1}^{n} i_{k-3,j}e_{k-1,j}e_{k,j}$. These $k-2$ FOFs ensure that the settings of variables in each block is the same if there is a weight $2k-3$ satisfying assignment to the $2k-3$ blocks of n variables.

The widget A_k can be drawn as follows.

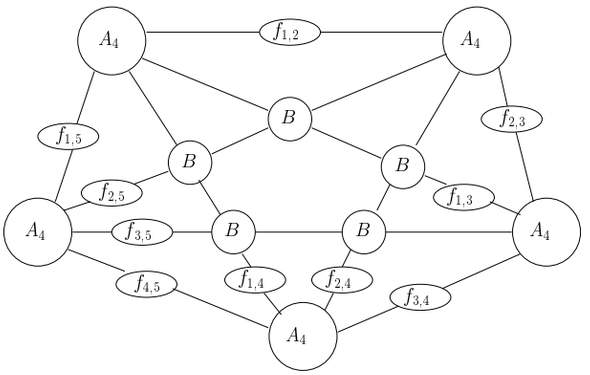


Since each internal block is connected to exactly two FOFs, the incidence graph for this widget can be drawn on the plane without crossing any edges.

The second widget removes crossover edges from the first stage of the construction. In the first stage, crossovers can occur in the incidence graphs because two FOFs may cross from one block to another. To eliminate this, consider each edge i, j in K_k with $i < j$ as a directed edge from i to j. In the construction, we send a copy of block i to block j. At each crossover point from the direction of block $u = [u_1, \ldots, u_n]$ and $v = [v_1, \ldots, v_n]$, insert a widget B that introduces two new blocks of n variables $u_1 = [u_{1_1} \ldots u_{1_n}]$ and $v_1 = [v_{1_1} \ldots v_{1_n}]$ and a FOF $f_B = \bigvee_{j=1}^{n} \bigvee_{l=1}^{n} u_j u_{1_j} v_l v_{1_l}$. The FOF f_B ensures that u_1 and v_1 are copies of u and v. Moreover, notice that the incidence graph for the widget B is also planar.

To complete the construction, we replace each of the original k blocks of n variables from the first stage with a copy of the widget A_{k-1}. At each crossover point in the graph, we introduce a copy of widget B. Finally, for each directed edge between blocks (i, j), we insert the original FOF $f_{i,j}$ between the last widget B and the destination widget A_{k-1}. Since one of the new blocks of variables created by the widget B is a copy of block i, the effect of the FOF $f_{i,j}$ in this new collections is the same as before.

The following diagram shows the full construction when $k = 5$.



Since each the incidence graph of each widget in this drawing is planar, the entire collection C of first order formulas has a planar incidence graph.

Now, if we assume that there are $c(k) = O(k^4)$ crossover points in standard drawing of K_k, then our collection has $c(k)$ B widgets. Since each B widget introduces two new blocks of n variables, this gives $2c(k)$ new blocks. Since we have k A_{k-1} widgets, each of which has $2(k-1) - 3 = 2k - 5$ blocks of n variables, this gives an additional $k(2k-5)$ blocks. So, in total, our construction has $f(k) = 2c(k) + 2k^2 - 5k = O(k^4)$ blocks of n variables. Note also that there are $g(k) = k(k-1)/2 + k(k-2) + c(k) = O(k^4)$ FOFs in the collection C.

As shown in our construction C has a weight $f(k)$ satisfying assignment (i.e., each block has exactly one variable set to true) if and only if the original

graph G has a clique of size k. Since the incidence graph of C is planar and each minterm in each FOF contains at most four variables, it follows that this construction is a parameterized reduction as claimed. □

In a similar manner the other two planar logic problems defined by Khanna and Motwani can be shown to be $W[1]$-hard. PTAS's for these problems therefore can never be useful, since the goodness of the approximation must be paid for in the exponent of the polynomial running time. A PTAS result alone establishes that an approximation problem is in the parameterized complexity class XP. By analogy, one would not reasonably claim any practical significance for a demonstration that a problem just belongs to *NP*. It would be interesting to sort out which problems with PTAS's have any hope of practical approximation, and for which such "good news" (see [3.7] for a comprehensive survey) is chimerical.

3.5 The Extremal Connection: A General Method Relating *FPT*, Polynomial-Time Approximation, and Pre-Processing Based Heuristics

The toolkit for establishing fixed-parameter tractability includes a number of mathematically deep methods: well-quasi-ordering, color-coding, and bounded treewidth — as well as the elementary methods of kernelization and search trees. Some of these positive methods are very powerful at *classifying* problems as fixed-parameter tractable, but are far from any practical significance (for example, methods based on well-quasiordering). For the purposes of practical algorithm design, *reduction to a problem kernel* is probably the single most important contribution to the systematic design of heuristics. This is in some sense a comprehensive connection, since *fixed-parameter tractability* is equivalent to *kernelizability* as shown by Lemma 3.3.1

There are several points to be noted about kernelization that lead to important research directions:

(1) Kernelization rules are frequently surprising in character, laborious to prove, and nontrivial to discover. Once found, they are small gems of *data reduction* that remain permanently in the heuristic design file for hard problems. No one concerned with any application of Hitting Set on real data should henceforth neglect Weihe's data reduction rules for this problem. The kernelization for Vertex Cover to graphs of minimum degree four, for another example, includes the following nontrivial transformation [3.24]. Suppose G has a vertex x of degree three that has three mutually nonadjacent neighbors a, b, c. Then G can be simplified by: (1) deleting x, (2) adding edges from c to all the vertices in $N(a)$, (3) adding edges from a to all the vertices in $N(b)$, (3) adding edges from b to all the vertices in $N(c)$, and (4) adding the edges ab and bc. Note that

this transformation is not even symmetric! The resulting (smaller) graph G' has a vertex cover of size k if and only if G has a vertex cover of size k. Moreover, an optimal or good approximate solution for G' lifts constructively to an optimal or good approximate solution for G. The research direction this points to is **to discover these gems of smart preprocessing for all of the hard problems**. There is absolutely nothing to be lost in smart pre-processing, no matter what the subsequent phases of the algorithm (even if the next phase is genetic algorithms or simulated annealing).

(2) Kernelization rules cascade in ways that are surprising, unpredictable in advance, and often quite powerful. Finding a rich set of reduction rules for a hard problem may allow the synergistic cascading of the pre-processing rules to "wrap around" hidden structural aspects of real input distributions. Weihe's train problem provides an excellent example. According to the experience of Alber, Gramm and Niedermeier with implementations of kernelization-based *FPT* algorithms [3.3], the effort to kernelize is amply rewarded by the subsequently exponentially smaller search tree. Similar results have also been reported by Moret *et al.* with respect to the Breakpoint Phylogeny problem [3.40].

(3) Kernelization is an intrinsically robust algorithmic strategy. Frequently we design algorithms for "pure" combinatorial problems that are not quite like that in practice, because the modeling is only approximate, the inputs are "dirty", etc. For example, what becomes of our Vertex Cover algorithm if a limited number of edges uv in the graph are *special*, in that it is forbidden to include *both* u and v in the vertex cover? Because they are local in character, the usual kernelization rules are easily adapted to this situation.

(4) Kernelization rules normally preserve all of the information necessary for optimal or approximate solutions. For example, Weihe's kernelization rules for the train problem (Hitting Set) transform the original instance G into a problem kernel G' that can be solved optimally, and the optimal solution for G' "lifts" to an optimal solution for G.

The importance of pre-processing in heuristic design is not a new idea. Cheeseman *et al.* have previously pointed to its importance in the context of artificial intelligence algorithms [3.15]. What parameterized complexity contributes is a richer theoretical context for this basic element of practical algorithm design. Further research directions include potential methods for mechanizing the discovery and/or verification of reduction rules, and data structures and implementation strategies for efficient kernelization preprocessing.

Lemma 3.3.1 tells us that a parameterized problem is fixed-parameter tractable if and only if there is a polynomial-time kernelization algorithm transforming the input (x, k) into (x', k') where $k' \leq k$ and $|x'| \leq g(k')$ for

some function g special to the problem. The basic schema is that reduction rules are applied until an *irreducible* instance (x', k') is obtained. At this point a *Kernel Lemma* is invoked to decide all those reduced instances x' that are larger than $g(k')$ for the kernel-bounding function g. For example, in the cases of Vertex Cover and Planar Dominating Set, if a reduced graph G' is larger than $g(k')$ then (G', k') is a no-instance. In the case of Max Leaf Spanning Tree large reduced instances are automatically yes-instances. (It is notable that for all three of these problems *linear kernelization*, $g(k) = O(k)$, has been established, in all cases nontrivially [3.17, 3.26, 3.2].)

> **How does one proceed to discover an adequate set of reduction rules, or elucidate (and prove) a bounding function $g(k)$ that insures for instances larger than this bound, that the question can be answered directly?**

The technique of *coordinatized kernelization* is aimed at these difficulties, and we will illustrate it by example with the Max Leaf Spanning Tree problem. Our objective is to prove:

The Kernel Lemma. If $(G = (V, E), k)$ is a reduced instance of Max Leaf Spanning Tree and G has more than $g(k)$ vertices, then (G, k) is a yes-instance.

We will prove the Kernel Lemma as a corollary to the following.

The Boundary Lemma. If $G = (V, E)$ is a reduced instance of Max Leaf Spanning Tree that is a yes-instance for k and a no-instance for $k+1$, then G has at most $h(k)$ vertices.

Let us first verify that the Kernel Lemma follows from the Boundary Lemma. We will make the mild assumption that our functions $g(k)$ and $h(k)$ are nondecreasing. Take $g(k) = h(k)$. Suppose (G, k) is a counterexample to the Kernel Lemma. Then G is reduced, and has more than $h(k)$ vertices, but is a no-instance, that is, G does not have a spanning tree with at least k leaves. Let $k' < k$ be the maximum number of leaves in a spanning tree of G. Then G is a yes-instance for k' and a no-instance for $k' + 1$. Since $k' < k$ and h is non-decreasing, G has more than $h(k')$ vertices, but this contradicts the Boundary Lemma.

The form of the Boundary Lemma (... which still needs to be proved, and we still need to discover what we mean by "reduced", and we also need to identify the particular bounding function h ...) is conducive to an *extremal theorem* style of argument based on a list of inductive priorities. The proof is sketched as follows.

Sketch Proof of the Boundary Lemma. The proof is by *minimum counterexample*. If there is any counterexample, then we can take G to be one that satisfies:

(1) G is reduced.
(2) G is connected and has more than $h(k)$ vertices.
(3) G is a no-instance for $k + 1$.

(4) G is a yes-instance for k, as witnessed by an t-rooted tree subgraph T of G that has k leaves. (We do not assume that T is spanning. Note that if T has k leaves then it can be extended to a spanning tree with at least as many leaves.)
(5) G is a counterexample where T has a minimum possible number of vertices.
(6) Among all of the G, T satisfying (1)–(5), T has a maximum possible number of internal vertices that are adjacent to a leaf of T.
(7) Among all of the G, T satisfying (1)–(6), the quantity $\sum_{l \in L} d(t, l)$ is minimized, where L is the set of leaves of T and $d(t, l)$ is the distance in T to the "root" vertex t.

Then we argue for a contradiction.

Comment. The point of all this is to set up a framework for argument that will allow us to see what reduction rules are needed, and what $g(k)$ can be achieved. In essence we are setting up a (possibly elaborate, in the spirit of extremal graph theory) argument by minimum counterexample — and using this as a discovery process for the *FPT* algorithm design. The witness structure T of condition (4) gives us a way of "coordinatizing" the situation — giving us some structure to work with in our inductive argument. How this strucuture is used will become clear as we proceed.

We refer to the vertices of $V - T$ as *outsiders*. The following structural claims are easily established. The first five claims are enforced by condition (3), that is, if any of these conditions did not hold, then we could extend T to a tree T' having one more leaf.

Claim 1: No outsider is adjacent to an internal vertex of T.
Claim 2: No leaf of T can be adjacent to two outsiders.
Claim 3: No outsider has three or more outsider neighbors.
Claim 4: No outsider with 2 outsider neighbors is connected to a leaf of T.
Claim 5: The graph induced by the outsider vertices has no cycles.

It follows from Claims (1)–(5) that the subgraph induced by the outsiders consists of a collection of paths, where the internal vertices of the paths have degree two in G. Since we are ultimately attempting to bound the size of G, this suggests (as a discovery process) the following reduction rule for kernelization.

Kernelization Rule 1: If (G, k) has two adjacent vertices u and v of degree two, then:

(Rule 1.1) If uv is a bridge, then contract uv to obtain G' and let $k' = k$.
(Rule 1.2) If uv is not a bridge, then delete the edge uv to obtain G' and let $k' = k$.

The soundness of this reduction rule is not completely obvious, although not difficult. Having now partly clarified condition (1), we can continue the

argument. The components of the subgraph induced by the outsiders must consist of paths having either one, two, or three vertices.

Because we are trying to efficiently bound the total number of outsiders (as well as everything else, eventually, in order to obtain the best possible kernelization bound $h(k)$), the situation suggests we should look for further reduction rules to address the remaining possible situations with respect to the outsiders. This discovery process leads us to the following further kernelization rules.

Kernelization Rule 2: If (G, k) is a (connected) instance of Max Leaf where G has a vertex u of degree one, with neighbor v, and where $\exists x \notin N(v)$ (that is, not every vertex of G is a neighbor of v), then transform (G, k) into (G', k'), where $k = k'$ and G' is obtained by:

(1) deleting u, and
(2) adding edges to make $N[v]$ into a clique.

The reader can verify that this rule is sound: (G, k) is a yes-instance if and only if (G', k') is a yes-instance.

Kernelization Rule 3: If (G, k) is a (connected) instance of Max Leaf where G has two vertices u and v such that either:

(1) u and v are adjacent, and $N[u] = N[v]$, or
(2) u and v are not adjacent, and $N(u) = N(v)$,

and also (in either case) there is at least one vertex of G not in $N[u] \cup N[v]$, then transform (G, k) to (G', k') where $k' = k - 1$ and G' is obtained by deleting u.

Returning to our consideration of the outsiders, we are now in the situation that for a reduced graph, the only possibilities are:

(1) A component of the outsider graph is a single vertex having at least two leaf neighbors in T.
(2) A component of the outsider graph is a K_2 having at least three leaf neighbors in T.
(3) A component of the outsider is a path of three vertices P_3 having at least four leaf neighbors in T.

The weakest of the ratios is given by case (3). We can conclude that the number of outsiders is bounded by $3k/4$.

The next step is to study the tree T. Since it has k leaves, it has at most $k - 2$ branch vertices. Using conditions (5) and (6), but omitting the details, it is argued that: (1) the paths in T between a leaf and its parental branch vertex has no subdivisions, and (2) any other path in T between branch vertices has at most three subdivisions (with respect to T). These statements are proved by various further structural claims (as in the analysis of the outsider population) that must hold, else one of the inductive priorities would fail (constructively) — a tree with $k + 1$ leaves would be possible, or a

smaller T, or a T with more internal vertices adjacent to leaves can be devised, or one with a better score on the sum-of-distances priority (7). Consequently T has at most $5k$ vertices, unless there is a contradiction. Together with the bound on the outsiders in a reduced graph, this yields a $g(k)$ of $5.75k$. □

The above sketch illustrates how the project of proving an FPT kernelization bound is integrated with the search for efficient kernelization rules. But there is more to the story. The argument above also leads directly to a constant-factor polynomial-time approximation algorithm in the following way. First, reduce G using the kernelization rules. It is easy to verify that the rules are approximation-preserving. Thus, we might as well suppose that G is reduced to begin with. Now take *any* tree T (not necessarily spanning) in G. If all of the structural claims hold, then (by our arguments above) the tree T must have at least n/c leaves for $c = 5.75$, and therefore we already have (trivially) a c-approximation. (It would require further arguments, but probably the approximation factor is much better than c.) If at least one of the structural claims does not hold, then the tree T can be improved against one of the inductive priorities. Notice that each claim is proved (in the kernelization argument above) by a constructive consequence. For example, if Claim 1 did not hold, then we can find a tree T' (by modifying T) that has one more leaf. Similarly, each claim violation yields a constructive consequence against one of the inductive priorities in the extremal argument for the kernelization bound. These consequences can be applied to our original T (and its successors) only a polynomial number of times (determined by the list of inductive priorities) until we arrive at a tree T' for which all of the various structural claims hold. At that point, we must have a c-approximate solution.

References

3.1 K. Abrahamson, R. Downey, and M. Fellows. Fixed parameter tractability and completeness IV: on completeness for $W[P]$ and `PSPACE` analogs. *Annals of Pure and Applied Logic* 73:235–276, 1995.

3.2 J. Alber, M. Fellows, and R. Niedermeier. Efficient data reduction for dominating set: a linear problem kernel for the planar case. To appear in the *Proceedings of Scandinavian Workshop on Algorithms and Theory (SWAT'02)*. Springer Lecture Notes in Computer Science, 2002.

3.3 J. Alber, J. Gramm, and R. Niedermeier. Faster exact algorithms for hard problems: a parameterized point of view. *Discrete Mathematics* 229:3–27, 2001.

3.4 S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science (FOCS'96)*, pages 2–11, 1996.

3.5 S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 38th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'97)*, pages 554–563, 1997.

3.6 V. Arvind, M. R. Fellows, M. Mahajan, V. Raman, S. S. Rao, F. A. Rosamond, and C. R. Subramanian. Parametric duality and fixed parameter tractability. Manuscript, 2001.

3.7 G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer-Verlag, Heidelberg, 1999.

3.8 N. Bansal and V. Raman. Upper bounds for MaxSat: further improved. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC'99)*. Springer Lecture Notes in Computer Science 1741, pages 247–258, 1999.

3.9 C. Bazgan. Schémas d'approximation et complexité paramétrée. Rapport de stage de DEA d'Informatique à Orsay, 1995.

3.10 M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In S. Miyano and T. Tagaki, editors, *Genome Informatics 1997*, Universal Academy Press, Tokyo, 1997, pages 25–34.

3.11 L. Cai. The complexity of coloring parameterized graphs. To appear in *Discrete Applied Mathematics*.

3.12 L. Cai, M. Fellows, D. Juedes, and F. Rosamond. Efficient polynomial-time approximation schemes for problems on planar structures: upper and lower bounds. Manuscript, 2001.

3.13 M. Cesati and L. Trevisan. On the efficiency of polynomial time approximation schemes. *Information Processing Letters* 64:165–171, 1997.

3.14 M. Cesati and H. T. Wareham. Parameterized complexity analysis in robot motion planning. In *Proceedings of the 25th IEEE International Conference on Systems, Man and Cybernetics: Volume 1*. IEEE Press, Los Alamitos, CA, pages 880–885, 1995.

3.15 P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 331-337, 1991.

3.16 C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. Manuscript, 2000.

3.17 J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'99)*. Springer Lecture Notes in Computer Science 1665, pages 313–324, 1999.

3.18 J. Chen and A. Miranda. A polynomial-time approximation scheme for general multiprocessor scheduling. In *Proceedings of the 31st Annual ACM Symposium on the Theory of Computing (STOC'99)*, pages 418–427, 1999.

3.19 F. Dehne, A. Rau-Chaplin, U. Stege, and P. Taillon. Solving large FPT problems on coarse grained parallel machines. Manuscript, 2001.

3.20 X. Deng, H. Feng, P. Zhang, and H. Zhu. A polynomial time approximation scheme for minimizing total completion time of unbounded batch scheduling. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC'01)*. Springer Lecture Notes in Computer Science 2223, pages 26–35, 2001.

3.21 R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, Heidelberg, 1998.

3.22 R. G. Downey and M. Fellows. Parameterized complexity after almost ten years: review and open questions. In *Proceedings of Combinatorics, Computation and Logic, DMTCS'99 and CATS'99*, Australian Computer Science Communications, Springer-Verlag, Singapore, vol. 21, pages 1–33, 1999.

3.23 R. G. Downey, M. Fellows, and U. Taylor. The parameterized complexity of relational database queries and an improved characterization of $W[1]$. In *Combinatorics, Complexity and Logic: Proceedings of DMTCS'96*. Springer-Verlag, Heidelberg, pages 194–213, 1997.

3.24 R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: a framework for systematically confronting computational intractability. In R. Graham, J. Kratochvíl, J. Nesetril, and F. Roberts, editors, *Proceedings of the DIMACS-DIMATIA Workshop*, Prague, 1997. AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 49, pages 49–99, 1999.

3.25 T. Erlebach, K. Jansen, and E. Seidel. Polynomial time approximation schemes for geometric graphs. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms (SODA'01)*, pages 671–679, 2001.

3.26 M. Fellows, C. McCartin, F. Rosamond, and U. Stege. Trees with few and many leaves. Manuscript, full version of the paper: Coordinatized kernels and catalytic reductions: an improved FPT algorithm for max leaf spanning tree and other problems. In *Proceedings of the 20th FSTTCS Conference*. Springer Lecture Notes in Computer Science 1974, pages 240–251, 2000.

3.27 J. Felsenstein. Private communication, 1997.

3.28 M. Galota, C. Glasser, S. Reith, and H. Vollmer. A polynomial time approximation scheme for base station positioning in UMTS networks. In *Proceedings of Discrete Algorithms and Methods for Mobile Computing and Communication*, 2001.

3.29 M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of* NP*-Completeness*. W. H. Freeman, San Francisco, 1979.

3.30 G. Gottlob, F. Scarcello, and M. Sideri. Fixed parameter complexity in AI and nonmonotonic reasoning. To appear in *The Artificial Intelligence Journal*. Conference version in *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, Springer Lecture Notes in Artificial Intelligence 1730, pages 1–18, 1999.

3.31 J. Gramm and R. Niedermeier. Faster exact algorithms for Max2Sat. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*. Springer Lecture Notes in Computer Science 1767, pages 174–186, 2000.

3.32 M. Grohe. Generalized model-checking problems for first-order logic. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS'01)*. Springer Lecture Notes in Computer Science 2010, pages 12–26, 2001.

3.33 M. Grohe. The parameterized complexity of database queries. In *Proceedings of the 20th ACM symposium on Principles of Database Systems (PODS'01)*, ACM Press, pages 82–92, 2001.

3.34 M. Hallett, G. Gonnett, and U. Stege. Vertex cover revisited: a hybrid algorithm of theory and heuristic. Manuscript, 1998.

3.35 F. Henglein and H. G. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 119–130, 1991.

3.36 Y. Karuno and H. Nagamochi. A polynomial time approximation scheme for the multi-vehicle scheduling problem on a path with release and handling times. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC'01)*. Springer Lecture Notes in Computer Science 2223, 36–47, 2001.

3.37 S. Khanna and R. Motwani. Towards a syntactic characterization of PTAS. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC'96)*, pages 329–337, 1996.

3.38 S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. In *Proceedings of the 6th Annual International Computing and Combinatorics Conference (COCOON'00)*. Springer Lecture Notes in Computer Science 1858, pages 137–147, 2000.

3.39 O. Lichtenstein and A. Pneuli. Checking that finite-state concurrents programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107, 1985.

3.40 B. Moret, S. Wyman, D. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium on Biocomputing (PSB'01)*, pages 583–594, 2001.

3.41 P. Moscato. Controllability, parameterized complexity, and the systematic design of evolutionary algorithms. Manuscript, 2001. See `http://www.densis.fee.unicamp.br/~moscato`

3.42 R. Niedermeier. Some prospects for efficient fixed-parameter algorithms. In *Proceedings of the 25th Conference on Current Trends in Theory and Practice oof Informatics (SOFSEM'98)*. Springer Lecture Notes in Computer Science 1521, pages 168–185, 1998.

3.43 R. Niedermeier and P. Rossmanith. An efficient fixed parameter algorithm for 3-hitting set. *Journal of Discrete Algorithms* 2(1), 2001.

3.44 C. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS'97)*, pages 12–19, 1997.

3.45 I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. *Electronic Colloquium on Computational Complexity Technical Report* 98-071, `http://www.ecc.uni-trier.de/eccc`.

3.46 V. Raman. Parameterized complexity. In *Proceedings of the 7th National Seminar on Theoretical Computer Science*, Chennai, India, pages 1–18, 1997.

3.47 H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. In *Proceedings of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX'00)*. Springer Lecture Notes in Computer Science 1913, pages 144–154, 2000.

3.48 R. Shamir and D. Tzur. The maximum subforest problem: approximation and exact algorithms. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pages 394–399, 1998.

3.49 U. Stege. Resolving conflicts in problems in computational biochemistry. Ph.D. dissertation, ETH, 2000.

3.50 M. Truszczynski. On Computing Large and Small Stable Models. In *Proceedings of the International Conference on Logic Programming*, pages 169–183, 1999. Full version to appear in *Journal of Logic Programming*.

3.51 K. Weihe. Covering trains by stations, or the power of data reduction. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experiments (ALENEX'98)*. Springer Lecture Notes in Computer Science 1619, pages 1–8, 1998.

3.52 K. Weihe. On the differences between practical and applied. Dagstuhl Workshop on Experimental Algorithmics, September 2000.

3.53 B. Wu, G. Lancia, V. Bafna, K-M. Chao, R. Ravi, and C. Tang. A polynomial time approximation scheme for minimum routing cost spanning trees. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pages 21–32, 1998.

4. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation

Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen

Department of Computer Science & Engineering
University of Washington, Box 352350, Seattle, WA 98195, USA
ladner@cs.washington.edu

Summary.

An experimental comparison of cache aware and cache oblivious static search tree algorithms is presented. Both cache aware and cache oblivious algorithms outperform classic binary search on large data sets because of their better utilization of cache memory. Cache aware algorithms with implicit pointers perform best overall, but cache oblivious algorithms do almost as well and do not have to be tuned to the memory block size as cache aware algorithms require. Program instrumentation techniques are used to compare the cache misses and instruction counts for implementations of these algorithms.

4.1 Introduction

The performance of an algorithm when implemented is a function of many factors: its theoretical asymptotic performance, the programming language chosen, choice of data structures, the configuration of the target machine, and many other factors. One factor that is becoming more and more important is how well the algorithm takes advantage of the memory hierarchy, its *memory performance*. Data to be processed by the algorithm can stored in different levels of the memory hierarchy: the registers on the processor chip, first level cache, second level cache, main memory, and secondary memory on disk. Each successive level of the memory hierarchy is slower and larger than the preceding level. When a datum is required by the processor it must be transferred from its current location in the hierarchy to the processor. Because of the time delay in moving the datum to the processor, typically surrounding data is also transferred down the memory hierarchy in a *block* that contains the required datum. This block transfer amortizes the transfer time of all the data in the hope that not just the one datum is required, but that surrounding data will be required soon. Typical block sizes are 1024 bytes from disk to main memory, 32 bytes from main memory to the level two and level one caches, and four bytes from the level one cache to the registers.

Most algorithms are not designed with memory performance in mind and probably shouldn't be. However, there are cases where an algorithm is in the "inner loop" where good memory performance is necessary. In these cases

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 78–92, 2002.
© Springer-Verlag Berlin Heidelberg 2002

designing for good memory performance is needed to achieve optimal performance. It is not difficult to find examples where a "memory sensitive" main memory algorithm can achieve a 50% reduction in running time over a similar "memory insensitive" algorithm. The reduction in running time can be attributed to the reduction in level two cache misses, where a cache miss is an access to a datum that is in main memory but not in the level two cache.

In this paper we concentrate on the memory performance of algorithms where the data resides in main memory and not in secondary memory. In particular, we examine the classic technique of *binary search*, an algorithm to locate an item among a static set of items. Classic binary search is so well known that it does not need any introduction. However, a quick analysis shows that its memory performance is poor. Suppose several items can fit into a memory block. In classic binary search the items are stored in a sorted array. The query item is compared with the middle item in the array. If it is equal, the search is completed. If it is smaller, the subarray to the left of the middle is searched in the same way. If it is larger, the subarray to the right is searched in the same way. The important point is that in the two latter cases the next item accessed is likely to be far from the middle of the array, so it is not in the same memory block. Thus, memory blocks are poorly utilized in classic binary search. Can binary search's memory performance be improved? The answer is a resounding yes, and there are several strategies to do so.

In this paper we examine two strategies for improving the memory performance of binary search. The first is the *cache aware* approach where items that are accessed together are stored together in the same memory block. Knowledge of the memory block size is needed to accomplish this. In this approach the items can be stored without the use of explicit pointers, but the layout of the items in memory does not constitute a sorted array. A disadvantage of cache aware search is that, because the items are organized into memory blocks, the algorithm does not achieve the perfect binary splitting into equal size subproblems. Cache aware algorithms have been studied in a number of different contexts [4.7, 4.8, 4.5, 4.10].

The second approach to improving the memory performance of binary search is the *cache oblivious* [4.9, 4.4, 4.1, 4.2, 4.3] approach where the items are organized in a universal fashion so that items that are accessed closely in time are stored near each other. The method is called cache oblivious because knowledge of the memory block size is not needed to achieve the organization. The advantage of the cache oblivious approach is that the organization of the data yields good memory performance at all levels of the memory hierarchy. One disadvantage of the cache oblivious approach is that it might not perform as well as the cache aware approach because it cannot take advantage of knowledge of the memory block size. Another disadvantage of the cache oblivious approach is that, although items can be accessed without explicit pointers, the computation to find the next item may be prohibitively expensive. This means that explicit pointers must be used, which increases

the memory footprint of the data structure, which may hurt memory performance.

We take an experimental approach in comparing cache aware and cache oblivious search. We first implemented in C classic binary search, cache aware search, and cache oblivious search. There are two versions of each implementation, one with implicit pointers and one with explicit pointers. As is normally done we did execution time studies on a wide range of data set sizes on several platforms. More interesting is our use of program instrumentation tools to count the number of machine instructions executed by each program and to simulate the number of cache misses that occurs for each implementation. The former metric is called *instruction count* and the latter metric is called *cache performance*. We simulated a direct mapped cache with several different memory block sizes.

We summarize our main results as:

1. In terms of execution time, both cache aware search with implicit pointers and cache oblivious search with explicit pointers perform comparably, and are both significantly faster the classic binary search.
2. Cache aware search with implicit pointers has slightly better cache performance than cache oblivious search with explicit pointers.
3. Cache aware search with implicit pointers has slightly worse instruction count performance than cache oblivious search with explicit pointers.

In summary, the cache oblivious approach is almost as effective as the cache aware approach to reducing cache misses for static search and has the advantage that it does not need to be tuned to the memory block size.

4.2 Organization

In Sections 4.3 and 4.4 we present cache aware search and cache oblivious search, respectively. In Section 4.5 we present the instrumentation tool ATOM [4.12] and how it is used for measuring cache misses and instruction counts. In Section 4.6 we present our experimental results. In Section 4.7 we present our conclusions.

4.3 Cache Aware Search

In this section, we present the cache aware approach for improving memory performance of binary search. The basic idea is to store in the same memory block those items that are most likely accessed together in time. In this way when an item moves from main memory to the cache, other items that are likely to be accessed soon are moved to the cache in the same block. Hence, cache misses are avoided. A simple way to achieve this is to use a k-ary tree

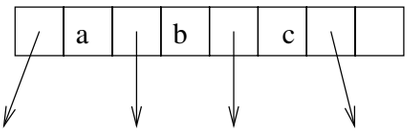


Fig. 4.1. Node of a cache aware 4-ary search tree stored in a 32 byte memory block

where a node contains $k-1$ items and k pointers to subtrees. To achieve the effect we want we choose k so that all the items and pointers fit in a memory block.

There are several ways to implement k-ary trees, one which employs explicit pointers and one that uses implicit pointers. In the former memory must be allocated to the pointers, while in the latter the address of the child of a node is calculated and no storage is wasted on pointers. Suppose that we know the cache line size is 32 bytes, and assume that an item and a pointer each occupies four bytes, we can store at most four pointers and three items. This means that our tree would be a 4-ary tree. Figure 4.1 depicts this example.

However, there are still four bytes in the memory block left unused. In order to make the node cache-align, these four bytes need to be padded in our structure. Hence, we lose some more memory for padding besides the memory used for pointers. A big disadvantage of the explicit pointer structure is the size of its memory footprint is increased by the inclusion of pointers and padding. An advantage of the explicit pointer structure is the speed in following pointers rather than calculating them.

Using implicit pointers helps to alleviate memory footprint problem. By not storing the explicit pointers, we can use the whole memory block to store keys, so the parameter k is larger. For example, for 32 byte memory block we now can store eight items instead of three in a memory block and have no padding. Interestingly, the utilization of the memory blocks for explicit and implicit pointers is about the same. If binary search is done within a node, then in the explicit pointer case two items and one pointer are touched most commonly. In the implicit pointer case three or four items are touched most commonly. The big win of implicit pointers is that the height of the tree, which bounds the number of cache misses, is much less.

For a k-ary tree, we layout the nodes in a contiguous piece of memory, starting from the root node going down, and from left to right for nodes at the same height. If the nodes are stored in an array, the root is stored at index 0 and the j-th child ($1 \leq j \leq k$) of the node stored at index i is stored at index $ik+(k-1)(j+1)$. This simple calculation replaces the explicit storage of pointers. The layout of the nodes of a 3-way cache aware search tree with implicit pointer is described in Figure 4.2.

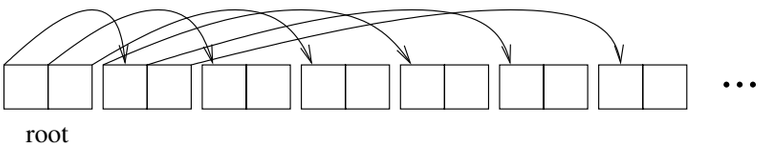


Fig. 4.2. Memory layout of a 3-way cache aware search tree with implicit pointers

4.4 Cache Oblivious Search

Cache oblivious algorithms operate under the same principles as cache aware algorithms. Both types of algorithms try to "cluster" data together in memory so that the locality of memory references is increased. The cache aware algorithm described above accomplishes this by "clustering" nodes of a binary search tree into nodes that fit into a memory block. The cache oblivious algorithm described by Prokop [4.9] approximates the same behavior, but does so without any knowledge of the cache parameters. Figure 4.3 shows how the cache oblivious algorithm lays out the data in memory to accomplish this.

Given a binary search tree of h (assuming h is a power of 2) levels, the memory layout algorithm works as follows. Cut the tree in half vertically, leaving one subtree above the cut and $2^{h/2}$ subtrees below the cut, giving a total of $2^{h/2}+1$ subtrees all of the same size. The top subtree is then placed

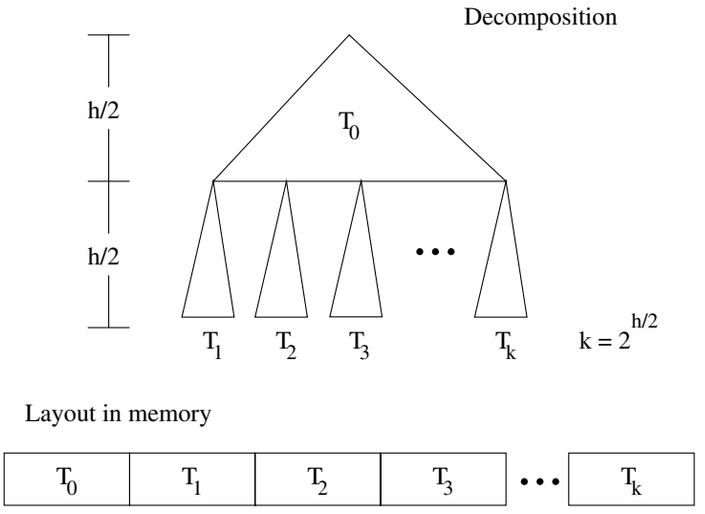


Fig. 4.3. Cache oblivious search tree decomposition and layout in memory

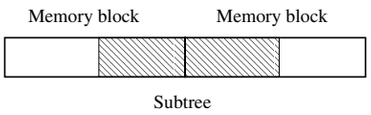


Fig. 4.4. A subtree of memory block size spans at most two memory blocks

in a contiguous block of memory followed by the $2^{h/2}$ subtrees from the left most to the right. The algorithm is then recursively applied to the to the top subtree followed by the bottom subtrees in left to right order. The algorithm terminates when it is applied to a subtree of one level, at which point it will add the single node into the array. As the algorithm recurses through each of the subtrees, it will eventually reach a tree which will occupy contiguous memory of size about the same as a memory block. This is similar to the behavior of the cache aware algorithm, the only difference in the two layouts is that the cache aware algorithm ensures that each "cluster" starts at the beginning, and spans only one memory block. The cache oblivious algorithm on the other hand cannot ensure that the cluster starts on a memory block boundary. Instead it guarantees that each cluster will span at most two memory blocks. This is illustrated below in Figure 4.4.

The cache aware algorithm knows the cache parameters and can therefore align the array in memory as to ensure that all the clusters are cache aligned. By virtue of the fact that the cache oblivious algorithm knows nothing of the cache parameters there is no way for it to ensure that a "cluster" does not begin somewhere in the middle of a memory block and thus ending in another memory block. Because of this fact, the cache aware algorithm will inherently have better cache performance than the cache oblivious algorithm. However, the cache oblivious algorithm does have the advantage that it does not have to be "hand tuned" for each cache size. Its properties ensure that each "cluster" will only span at most two memory blocks no matter the memory block size, where as the cache aware algorithm must be adjusted for each memory block size to ensure that its properties hold. As was the case with the cache aware implementation, the cache oblivious algorithm can be implemented with both explicit and implicit pointers. Implicit pointers have the benefit of reducing the memory footprint of a single node, and thus increase the overall cache performance. However the computation of the implicit pointers at run time impacts the instruction count of the algorithm and can have a negative effect on performance.

In recent work Bender et al. [4.2] and Brodal et al. [4.3] have used the cache oblivious static search tree as the basis of a cache oblivious dynamic search structures that allow for insertions and deletions. In particular, Brodal et al. have discovered a very elegant and efficient way to calculate the pointers in the cache oblivious static search tree with implicit pointers. For our cache oblivious search tree with implicit pointers we use a more compu-

tation intensive algorithm for computing pointers which is described in the next paragraph. Hence, our cache oblivious search tree with implicit pointers have high instruction counts and execution times. This could be remedied by using the Brodal et al. calculation of pointers.

As described in Figure 4.3, the layout of the cache oblivious search tree in memory is determined by recursively "cutting" the tree in half height-wise and placing the nodes in contiguous memory starting with the top half of the tree. It is not surprising that traversing the tree also involves recursively cutting the tree. The algorithm works as follows. Initially the algorithm begins it search at the root of the tree, the first level of the tree, and the initial "cut" is located at h, where h is the height of the tree. If the number of levels that separate the current node from the next cut is greater than or equal to two, a new cut is placed halfway between the current level of the search and the level of the next cut. If the current node is at the same level as the next cut an inter-cut traversal is done. Otherwise an intra-cut traversal is done. This process is repeated at the new node until the search succeeds or fails. An intra-cut traversal is defined as follows. Let i be the index of the current node in the search. If the difference between the level of the current node and the level of the next cut is ℓ, where $\ell \leq 2$ the left child of the current node is located at $i+1$ while the right child is located at $i+2^\ell$. An inter-cut traversal is done in the case that by moving to the next node in the traversal, we cross over an existing cut. In this case, the next node is located in memory after all the nodes above the cut and after all the nodes between the cut and the next cut in the subtrees to the left of the next node. If the number of levels in the tree above it is d and the number of trees to the left of it is s then j child $(1 \leq j \leq 2)$ of the node indexed at i is located at

$$(2s + j - 1)(2^\ell - 1) + 2^{d+1} - 1.$$

The quantity $2^{d+1} - 1$ is the number of nodes above the cut and the quantity $2^\ell - 1$ is the number of nodes in each of the subtrees to the left of the next node. There are either $2s$ or $2s + 1$ subtrees to the left of the next node depending on whether the traversal goes left or right respectively. A stack can be used to maintain the current cut by simulating the recursive construction described in Figure 4.3. The values of d and s can be maintained easily and the value of ℓ can be calculated from the cut value and d.

4.5 Program Instrumentation

Program instrumentation is a general way to understand what an executing program is doing. Program instrumentation can be achieved automatically or manually. For example, when a compiler is called with the debugger on, then the executable code is augmented automatically to allow the user to view values of variables or other quantities. The semantics of the program should be the same whether or not the debugger is turned on. Manual instrumentation

is where the programmer inserts instructions into the source code to measure some quantity or print out some intermediate values. For this study we employed the system ATOM [4.12] which enables the user to build customized tools to automatically instrument and analyze programs that run on DEC alphas. Other program instrumentation tools that are useful for measuring memory performance are Cacheprof [4.11] and Etch [4.6].

The programmer provides three pieces of code to ATOM: (i) the unlinked object code of the program to be instrumented, (ii) *instrumentation code* that tells atom what "sensors" to insert into the object code and where to place them, and (iii) *analysis code* that processes the sensor data from the executing program to provide information about the execution. ATOM takes the three pieces and produces an instrumented program executable. When the executable is then run, it has the same semantics as the uninstrumented program, but during the execution the sensors gather data that is processed by the analysis code. Figure 4.5 gives a picture of the ATOM system. A simple example of the use of ATOM is an *instruction counter*. The instrumentation code inserts an increment-counter instruction after every instruction in the object code. The analysis code sets the counter to zero initially and outputs the final count on termination. We employ such an instruction counter in our study.

A second, more sophisticated example, is a *trace driven cache simulator*. In this case the instrumentation code inserts instructions after each load and store to sense the memory address of the operand. The analysis code is a cache simulator that takes the address as input and simulates what a memory system would do with the address. In addition, the analysis code keeps track of the number of loads and stores to memory and how many accesses are misses. Figure 4.6 shows the instrumentation code for a cache simulator and Figure 4.7 shows the analysis code for a very simple one level, direct mapped

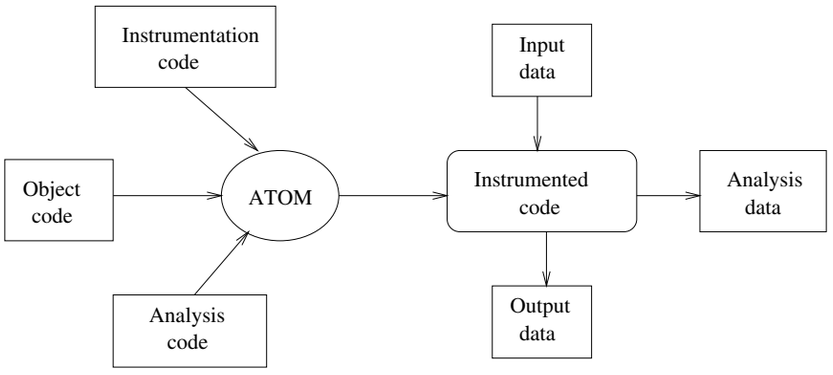


Fig. 4.5. Schematic of the ATOM system

```
Instrument() {
  Proc *p;  Block *b; Inst *i;
  AddCallProto("LoadReference(VALUE)");
  AddCallProto("StoreReference(VALUE)");
  AddCallProto("PrintResults()");
  for (p = GetFirstProc(); p != NULL;  p = GetNextProc(p)) {
    for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {
      for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i))  {
        if (GetInstClass(i) == ClassLoad ||
          GetInstClass(i) == ClassFload) {
            AddCallInst(i,InstBefore,"LoadReference",EffAddrValue);
          xs}
        if (GetInstClass(i)== ClassStore ||
          GetInstClass(i)== ClassFstore) {
            AddCallInst(i,InstBefore,"StoreReference",EffAddrValue);
          }
        }
      }
    }
  }
  AddCallProgram(ProgramAfter,"PrintResults");
}
```

Fig. 4.6. Instrumentation code for a trace driven cache simulator

cache simulator. We used a trace driven cache simulator similar to this one for our study.

In the instrumentation code, Figure 4.6, the nested for loops identify each procedure in the object code, then each basic block within the procedure, then each instruction within the basic block. If the instruction is a load, then code is inserted before the instruction which calls `LoadReference` in the analysis code passing `EffAddrValue`, the operand, as a parameter. If the instruction is a store, then code is inserted before the instruction which calls `StoreReference` in the analysis code passing `EffAddrValue`, the operand, as a parameter. At the end of the program code is inserted which calls `PrintResults`.

The analysis code implements a direct mapped cache with size `CACHE_SIZE` in bytes and block size `BLOCK_SHIFT` in bits. The analysis code maintains the array `tags` which stores the memory addresses that currently reside in the cache. In addition it maintains counters for the number of load and store references and load and store misses. For our study we use a similar cache simulator, but we add the load and store values into one value for both references and misses. We chose to simulate a one level cache because the cache miss penalty for the level two cache is typically much greater than that for the level one cache. Having just one number representing the cache performance is reasonable compromise considering that level one cache misses tend to have a low order effect on performance on large data sets.

ATOM is a powerful tool but it must be used properly to obtain accurate results. First, the analysis code is interleaved with the code to be analyzed. This means that the instrumented code can be considerably slower than the uninstrumented code. This is not a serious problem with either instruction

```
void generalreference(long address, int isLoad) {
  int index = (address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
  long tag = address >> (BLOCK_SHIFT + INDEX_SHIFT);
  int returnval;

  if (tags[index] != tag) {
     if (isLoad){
loadmisses++;
tags[index] = tag;
     }
     else  {
storemisses++
     }
  }
  if (isLoad) {
     loadreferences++;
  }
  else {
     storereferences++;
  }
}
void LoadReference(long address) {
   generalreference(address, 1);
}
void StoreReference(long address) {
   generalreference(address, 0);
}
```

Fig. 4.7. Analysis code for a trace driven cache simulator for a one level, direct mapped cache

counting or cache simulation because the analysis code is quite efficient. For cache simulation it is important not to use dynamic memory in the analysis code. Use of dynamic memory would cause a difference in the addresses used by the instrumented and uninstrumented codes, and distort the results. This is not a problem in our case because we used static memory to allocate the `tags` array and counters. Finally, the trace driven cache simulator is just that. It does not measure cache misses caused by swapping, TLB misses, or instruction cache misses.

4.6 Experimental Results

In order to better understand the alternative static search algorithms we implemented in C six algorithms: classic binary search, cache aware search, and cache oblivious search each with explicit and implicit pointers versions. All studies were for data sets the range from 128 to 2,097,152 and for larger data sets when possible. All items and pointers used are four bytes. In order to compare our static search algorithms we employed program instrumentation for trace driven cache simulation and instruction counts. These studies were done using ATOM on a Compaq Alpha 21164. In addition, we performed two execution studies one on Windows and one on Linux. In Table 4.6 we list the computer configurations and compilers. All the caches in the two platforms

Table 4.1. Computer configurations and compilers used in the execution time studies

	Windows	Linux
Operating System	Windows 2000 "Professional"	Linux Mandrake 7.2
Processor	533 MHz Intel Celeron	350 MHz Intel Pentium II
Memory	64 MB	128 MB
Memory Block	32 B	32 B
L2 Cache	128 KB	512 KB
L1 Cache	32 KB	32 KB
Compiler	MSVC 6.0	gcc 2.95.2
Options	Release Build Highest Option	-O3 (highest setting)

are 4-way set-associative and all block sizes are 32 bytes. In all the studies each data point represents the median of ten trials where a trial consisted of n random successful lookups where n is the number of items. The median of ten is computed as the average of the fifth and sixth ranked trials to avoid the effect of outliers. For a given n, the ten measured trials were preceded by n unmeasured successful lookups to warm up the cache.

Figure 4.8 gives the results a cache simulation using ATOM where we simulated a direct mapped cache of size 8,192 bytes and a memory block size of 32 bytes. In the x-axis we plot the number of items on a log scale and in the y-axis we plot the number of cache misses. We see that the cache aware search with implicit pointers has the fewest cache misses, while classic binary search has the most. All the algorithms that use implicit pointers have fewer cache misses than their explicit pointer counterparts showing the effect of the larger memory footprint for the explicit pointers. Most important is that both cache oblivious and cache aware search algorithms have much better memory performance than classic binary search.

Figure 4.9 gives the results of instruction counting using ATOM for the algorithms. In the x-axis we plot the number of items on a log scale and in the y-axis we plot the number of Compaq Alpha 21164 instructions executed per lookup. We see immediately the high price in instruction count that is paid for our version of cache oblivious search with implicit pointers. The instruction count penalties for implicit pointers for classic binary search and cache aware search are small. The explicit pointer versions of classic binary search and cache oblivious search execute the fewest instructions per lookup. Cache aware search with explicit pointers has slightly more instructions per lookup because it does not achieve perfect binary splitting into equal size subproblems.

In our execution time studies shown in Figures 4.10 and 4.11 in the x-axis we plot the number of items on a log scale and in the y-axis we plot the time per lookup measured in microseconds. Each trial was measured using `time.h` from the Standard C library.

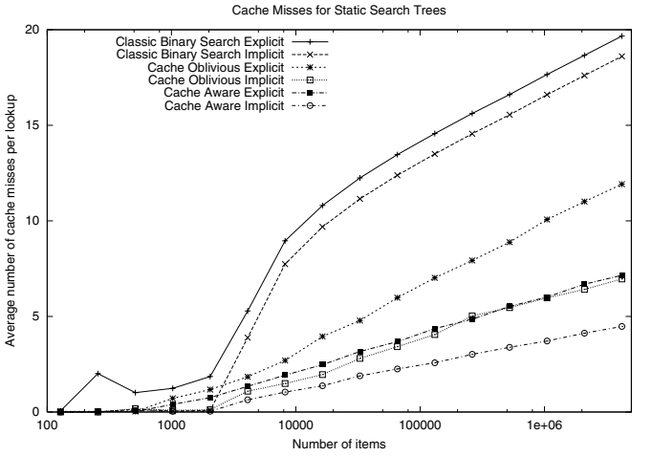


Fig. 4.8. Cache misses per lookup for static search algorithms

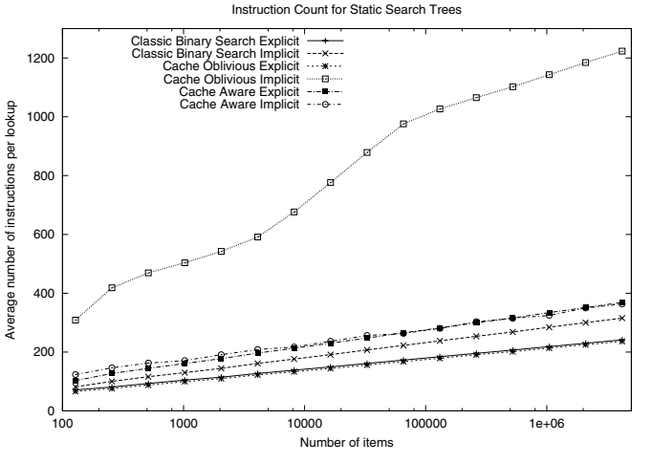


Fig. 4.9. Instruction count per lookup for static search algorithms

Figure 4.10 gives the results of an execution time study using Windows. Cache aware search with implicit pointers is the fastest, but cache oblivious search with explicit pointers is not far behind. Cache oblivious search with implicit pointers is the slowest of all because of the high cost of computing pointers.

Figure 4.11 gives the results of an execution time study using Linux. Again, cache aware search with implicit pointers is the fastest, but cache oblivious search with explicit pointers is not far behind. Again, cache oblivious search with implicit pointers is the slowest of all because of the high cost

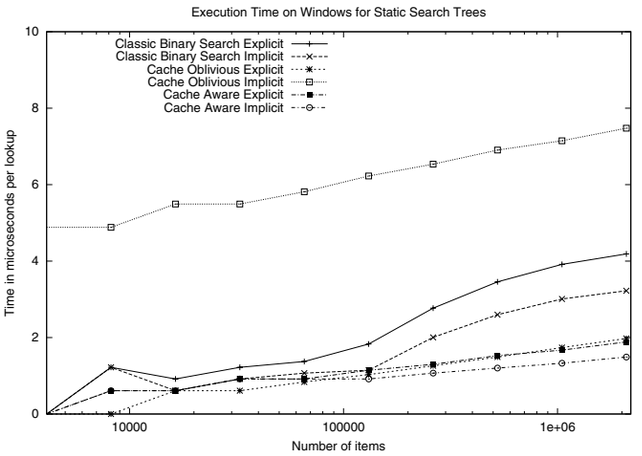


Fig. 4.10. Execution time on Windows for static search algorithms

of computing pointers. Inexplicably, cache aware search with explicit pointers showed consistently poor performance under Linux. We looked at a number of possible causes for the poor performance but were not able to pin down a reason for it. We believe that the Linux behavior perhaps demonstrates the perils of cache aware programming. The cache oblivious algorithms performed consistently on both platforms.

4.7 Conclusion

Both cache aware and cache oblivious search perform better than classic binary search on large data sets. Cache aware search algorithms have the disadvantage that they require knowledge of the memory block size. Cache oblivious search algorithms have only slightly worse memory performance than cache aware search, but in our study only the explicit pointer version of oblivious search has comparable overall performance. As mentioned earlier Brodal et al. [4.3] have found a way to compute the implicit pointers efficiently in the cache oblivious algorithm. The cache oblivious search algorithms do not require knowledge of the memory block size to achieve good memory performance. Finally, program instrumentation tools like ATOM let us obtain a deeper understanding of the performance of these algorithms.

Acknowledgments

The research was supported by NSF Grant No. CCR-9732828 and by Microsoft. Ray Fortna and Bao-Hoang Nguyen were undergraduate students at

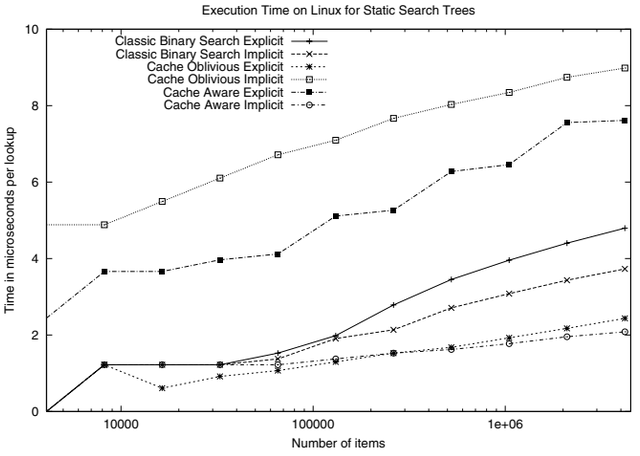


Fig. 4.11. Execution time on Linux for static search algorithms

the University of Washington at the time this paper was written. Ray Fortna was supported by a NSF REU.

Thanks to the anonymous referees for suggesting several improvements to the paper.

References

4.1 M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 399–409, 2000.

4.2 M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 29–38, 2002.

4.3 G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 39–48, 2002.

4.4 M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 285–297, 1999.

4.5 J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 78–89, 1999.

4.6 T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. The USENIX Windows NT Workshop, 1997. See `www.usenix.org/publications/library/proceedings/usenix-nt97/romer.html`.

4.7 A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, vol. 1, 1996.

4.8 A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms* 31:66–104, 1999.
4.9 H. Prokop. Cache-Oblivious Algorithms. Master's Thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.
4.10 S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 829–838, 2000.
4.11 J. Seward. Cacheprof. See `www.cacheprof.org`.
4.12 A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation (PLDI'94)*, pages 196–205, 1994.

5. Using Finite Experiments to Study Asymptotic Performance

Catherine McGeoch[1], Peter Sanders[2]*, Rudolf Fleischer[3], Paul R. Cohen[4], and Doina Precup[4]

[1] Amherst College, Amherst, MA, USA
ccm@cs.amherst.edu
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
sanders@mpi-sb.mpg.de
[3] The Hong Kong University of Science and Technology, Hong Kong
rudolf@cs.ust.hk
[4] University of Massachussetts, Amherst, MA
{dprecup,cohen}@cs.umass.edu

Summary.

In the analysis of algorithms we are interested in obtaining closed form expressions for algorithmic complexity, or at least asymptotic expressions in $\mathcal{O}(\cdot)$-notation. It is often possible to use experimental results to make significant progress towards this goal, although there are fundamental reasons why we cannot guarantee to obtain such expressions from experiments alone. This paper investigates two approaches relating to problems of developing theoretical analyses based on experimental data.

We first consider the scientific method, which views experimentation as part of a cycle alternating with theoretical analysis. This approach has been very successful in the natural sciences. Besides supplying preliminary ideas for theoretical analysis, experiments can test falsifiable hypotheses obtained by incomplete theoretical analysis. Asymptotic behavior can also sometimes be deduced from stronger hypotheses which have been induced from experiments. As long as complete mathematical analyses remains elusive, well tested hypotheses may have to take their place. Several examples are given where average complexity can be tested experimentally so that support for hypotheses is quite strong.

A second question is how to approach systematically the problem of inferring asymptotic bounds from experimental data. Five heuristic rules for "empirical curve bounding" are presented, ogether with analytical results guaranteeing correctness for certain families of functions. Experimental evaluations of the correctness and tightness of bounds obtained by the rules for several constructed functions and real datasets are described.

5.1 Introduction

The complexity analysis of algorithms is one of the core activities of computer scientists, especially in the branch of theoretical computer science known as algorithmics. The ultimate goal would be to find closed form expressions for the runtime (or other measures of resource consumption), in terms of

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 93–126, 2002.
© Springer-Verlag Berlin Heidelberg 2002

input parameters of interest. Since this is usually too complicated, we are often content with asymptotic expressions for the worst case performance depending on a small number of input parameters like problem size, which are usually presented in $\mathcal{O}(\cdot)$-notation. Even this task can be very difficult so it is important to use all available tools.

In this paper we investigate the empirical version of this primary activity – how to use finite experimental data to shed insight on universal asymptotic properties of algorithms. We illustrate both the promise and the difficulties inherent in the use of experiments to suggest, support, and refute hypotheses about asymptotic behavior. Experimental data can be employed for asymptotic analysis both indirectly – for example, in support of conjectures necessary to theoretical arguments; and directly, by extrapolation of trend data beyond the range of experimentation. In the latter scenario, we consider a specific problem, which we call *empirical curve-bounding*: given a set of data points (N_i, Y_i) obtained from an experiment in which $Y_i = f(N_i)$, for some unknown function $f(n)$, find complexity classes $O(g_u(n))$ and/or $\Omega(g_l(n))$ to which $f(n)$ belongs.

This paper has two goals. The first is to show how, with some care, it is possible to obtain good insights about asymptotic trends, based on analyses of data obtained from experiments. One way to make the meaning of "some care" more precise is to apply the terminology of the scientific method [5.31]. The scientific method views science as a cycle between theory and practice. Theory can inductively or (partially) deductively[1] formulate falsifiable hypotheses which can be tested by experiments. The results may then yield new or refined hypotheses. This mechanism is widely accepted in the natural sciences and is often viewed as a key to the success of these disciplines. We present four examples of ways in which the scientific method can be applied to the use of experimentation to advance the goals of asymptotic algorithm analysis, using problems in parallel disk scheduling, random polling, shellsort, and randomized process allocation.

The second goal is to evaluate a collection of curve-bounding techniques, in order to identify their practical limitations. Unfortunately, no data analysis method for inferring asymptotic trends in data can be guaranteed correct for all data sets: to see this, note that for any finite vector of problem sizes, there are functions of arbitrarily high degree that are indistinguishable from the constant function c at those problem sizes. Therefore any algorithm for this problem must be regarded as a heuristic that sometimes fails. We desire robust heuristics that produce correct bound estimates (or clear indications of failure) for broad classes of functions and for functions that tend to arise in practice.

We describe five simple heuristics (or rules) for curve bounding, and a hybrid rule that handles some specific pathologies. For each of the five rules,

[1] Inductive reasoning draws general conclusions from specific data whereas deductive reasoning draws specific conclusions from general statements.

we present analytical results guaranteeing correctness for certain families of functions. Then, using a variety of algorithmic data sets, we evaluate the rules in "typical" and in near-pathological situations. Negative results concerning two plausible rules that turned out to have high failure rates are also presented.

In our informal and designed experiments with little or no random noise in the data, all the rules generally provide correct asymptotic bounds that are within about a $\sqrt{n}$ factor of the true asymptotic bound. The reliability of the rules deteriorates, however, in the presence of random variation in the data, and/or when too-large constants or negative coefficients appear in second-order terms. Fortunately it is usually easy in algorithmic problems to reduce the noise problem by taking more experiments or applying variance reduction techniques during experimentation. It is of course possible to reduce the effect of large second-order terms by taking larger problem sizes, but the rules can be slow to respond to this type of change. A hybrid diagnostic method described in Section 5.6 can be used with success on such problems.

This explicit study of techniques for curve-bounding appears to be completely new. We can find no techniques in the statistical and data analysis literature specifically designed for finding asymptotic bounds on data, although much is known about fitting curves to data. As we shall demonstrate, good algorithms for curve fitting are not always best for curve bounding, and vice versa.

The importance of experiments in algorithm design and analysis has gained much attention in the past decade. New workshops (ALENEX, WAE) and journals (ACM Journal of Experimental Algorithmics) have been installed, and established conferences (e.g., SODA, ESA) explicitly call for experimental work. Several articles [5.4], [5.19], [5.27], [5.28]) present guidelines for performing experiments on algorithmic research problems, and one book [5.12] presents methods of data analysis in the context of experimentation on heuristic algorithms. Using the scientific method as a basis for algorithmics was proposed by Hooker [5.17], but similar ideas concerning experimental computer science in general can also be found in other papers [5.14, 5.15, 5.3, 5.37, 5.16, 5.23, 5.29, 5.41].

Section 5.2 reviews the main difficulties in experimental algorithmics and explains how to partially solve them. Section 5.3 gives several concrete examples of using experimental results to suggest, support, or to falsify hypotheses about algorithmic performance. The algorithms presented in this section are randomized, with expected resource consumption dependent only on input size so that many repeated experiments give us rather accurate information on average behavior. On the other hand, all the algorithms are nontrivial to analyze analytically. It turns out that in this situation the scientific method with a close, problem specific interaction between theoretical and experimental reasoning yields quite accurate insight on the asymptotic behavior of

the algorithm. For example, in Section 5.3.2 we are able to resolve even an additive $\mathcal{O}(\log \log n)$ term.

We then turn to a systematic evaluation of rules for the empirical curve-bounding problem. Section 5.4 presents each rule R, together with a "justification" that describes a class of functions for which the rule is guaranteed correct. Section 5.5 presents an empirical study of the rules using data sets from constructed parameterized functions. We observe that some rules are sensitive to large lower order terms and some to random noise, and some to both. Most of the rules are surprisingly unresponsive to changes in the largest problem size. One rule produces bounds that are rarely incorrect and rarely tight. A second collection of data comes from eight experimental studies of algorithms, to assess performance on "typical" algorithmic problems. In three cases there is at least a logarithmic gap in known analytical bounds, and we show how the rules can (and cannot) be used to support conjectures that tighten the gaps.

Section 5.4 assumes some familiarity with data analysis terms such as *correlation coefficient*, *least-squares regression*, and *residuals*, which may be found in any introductory statistics textbook. For introductions to the curve-fitting methods adapted here for curve-bounding, see Atkinson [5.1], Cohen [5.12], Chambers et al. [5.11], Rawlins [5.33], or Tukey [5.42]. Algorithms for domain-independent function finding [5.36] might be adapted to curve bounding but are not considered here.

Finally, Section 5.7 discusses the role of the scientific method in the context of experimental analysis of data and summarizes our observations about curve-bounding rules.

We emphasize that this work represents a small initial investigation of a potentially large research area. This paper only scratches the surface of a related important methodological topic, namely how to perform experiments on algorithms, and how to evaluate the confidence in our findings statistically. Our analyses are far from complete, and we do not consider here many interesting methodological and statistical questions, function classes, function parameters, rule variations, or multivariate problems.

In specific examples, we mostly consider cases where it is of interest to bound the complexity of algorithms for inputs of size n, using functions of the single parameter n. Later sections emphasizing data analysis use the symbol x in place of n, to refer to the "control parameter" in the experiment, but again we assume that only one such control parameter is present. Issues of experimentation with combinations of control parameters is outside the scope of this paper.

Of course, many problems in experimental evaluation include combinations of parameters (such as problem size n, graph density d, and algorithm tuning parameter p). But these problems can sometimes be studied by varying each parameter in turn while holding others fixed.

5.2 Difficulties with Experimentation

There is no question that experimental analysis of algorithms presents several fundamental problems to the researcher. Some of the major difficulties are surveyed in this section.

Too Many Inputs. Perhaps the most fundamental problem with algorithmic experimentation is that we can rarely test all possible inputs, even for bounded input size, because there are usually exponentially (or infinitely) many of them. In application-oriented research this problem may be mitigated by collections of test instances which are considered "typical".[2] For example, there is a large class of *oblivious* algorithms where the execution time only depends on a small number of parameters like the input size, for example, matrix multiplication. Although many oblivious algorithms are easy to analyze directly, experiments can sometimes help. Furthermore, there are algorithmic problems with few inputs. For example, the locality properties of several space filling curves were first found experimentally and then proven analytically. Later it turned out that a class of experiments can be systematically converted into theoretical results valid for arbitrary curve sizes [5.30].

But in most cases there are far too many instances to allow exhaustive testing. In these situations, our rich statistical understanding of random sampling makes algorithm randomization and average case analyses most important for experimentation. Randomization can be used to convert a hypothesis about "all instances" into one about behavior "on average," for which experimental approaches are most suited. For example, every sorting algorithm which is efficient on average can be transformed into an algorithm for worst-case instances by permuting the inputs randomly. In this case, a few hundred experimental trials with random inputs can give a reliable picture of the expected performance of the algorithm for inputs of a given size. On the other hand, closed form analyses of randomized algorithms can be very difficult to obtain. For example, the average performance of randomized Shellsort has been open for a long time [5.38]. Section 5.3.3 presents an experimental study of Shellsort.

Unbounded Input Size. Another problem with experiments is that we can only test a finite number of input *sizes*. As a result, no inference about asymptotic behavior is reliable. For example, assume we observe that some sorting algorithm needs an average of $C(n) \leq 3n \log n$ comparisons[3] for $n < 10^6$ elements. We cannot claim that $C(n) \leq 3n \log n$ as a theorem, since quadratic behavior might set in for $n > 42 \cdot 10^6$. Here, the scientific method partially saves the situation. We can formulate the hypothesis $C(n) \leq 3n \log n$, which is scientifically sound since it can be falsified by presenting an instance of size n with $C(n) > 3n \log n$.

[2] For example, a list with 23 collections of problem instances can be found under `http://mat.gsia.cmu.edu/Resources/Problem_Instances/`

[3] Throughout this paper $\log x$ stands for the base two logarithm $\log_2 x$.

Note that not every sound hypothesis is a good hypothesis. For example, we would be cowardly to change the above hypothesis to $C(n) \leq 100000n \log n$, since it would be difficult to falsify it even if it later turns out that the true bound is $C(n) = n \log n + 0.1n \log^2 n$. Issues like accuracy, simplicity, and generality of hypotheses also arise in the natural sciences and should not be obstacles to the use of the scientific method here.

$\mathcal{O}(\cdot)$-s are not Falsifiable. The next problem is that an asymptotic expression cannot be used directly in formulating a scientific hypothesis since it could never be falsified experimentally. For example, if we claim that a certain sorting algorithm needs at most $C(n) \in \mathcal{O}(n \log n)$ comparisons it cannot even be falsified by a set of inputs which clearly indicate quadratic behavior, since we could always claim that this quadratic development would stop for sufficiently large inputs. This problem can be solved by formulating a hypothesis which is stronger than the asymptotic expression we really have in mind. The hypothesis $C(n) \leq 3n \log n$ used above is a trivial example. A less trivial example is given in the study of Shellsort in Section 5.3.3.

Complexity of the Machine Model. Although the actual execution time of an algorithm is perhaps the most interesting subject of analysis, this measure of resource consumption is often difficult to model by closed form expressions. Caches, virtual memory, memory management, compilers, and interference from other processes all influence execution time in ways that are difficult to predict.[4] At some loss of accuracy, this problem can be solved by counting the number of times a certain set of source code operations (which cover all the inner loops of the program) is executed. This count often suffices to capture the asymptotic behavior of the code in a machine-independent way. For example, for comparison-based sorting algorithms it is usually sufficient to count the number of key comparisons.

Finding Hypotheses. Except in very simple cases, it is almost impossible to guess exactly an appropriate formula for a worst case performance, given only measurements, even when the investigated resource consumption only depends on input size. For example, the measured function may be non-monotonic but we are only interested in a monotonic upper bound. There are often considerable contributions of lower order terms for small inputs. Indeed our experience described in later sections shows that simple fitting methods sometimes just won't work, especially if we are interested in fine distinctions like logarithmic factors.

In some cases the scientific method can help to mitigate this difficulty by applying problem-specific information to the study. We may be able to handle a related or simplified version of the system analytically, or we can

[4] Remember that the above is also an argument *in favour* of doing experiments because the full complexity of the hardware is difficult to model theoretically. We only mention it as a problem in the current context of inducing asymptotic expressions from experiments.

make "heuristic" steps in a derivation of a theoretical bound. Although the result is not a theorem about the target system, it is good enough as a hypothesis about its behavior in the sense of the scientific method. Section 5.3 gives several examples of this powerful approach which so far seems to be underrepresented in algorithmics.

5.3 Promising Examples

Our first example in Section 5.3.1 can be viewed as the traditional use of experiments as a method to generate conjectures on the behavior of algorithms — but it has an additional interpretation in the sense that experiment plus theory (on a less attractive algorithm) yields a useful hypothesis. Section 5.3.2 gives an example in the same category but using a less well known approach. Rather than simplifying the algorithm, we simplify the analysis by making simplifying assumptions (independence) in the middle of the derivation. The resulting bound has the status of a theory in the sense of the scientific method and is then validated by simulation. Sections 5.3.3 and 5.3.4 touch on the difficult question of how to use experiments to learn something about the asymptotic complexity of an algorithm. Finally Section 5.3.4 is a good example how experiments can suggest that an analysis can be sharpened.

5.3.1 Theory with Simplifications: Writing to Parallel Disks

Consider the following algorithm, EAGER, for writing D randomly allocated blocks of data to D parallel disks. EAGER is an important ingredient of a general technique for scheduling parallel disks [5.35]. We maintain one queue Q_i for each disk. The queues share a buffer space of size $W \in \mathcal{O}(D)$. We first put all the blocks into the queues and then write one block from each nonempty queue. When the sum of the queue lengths exceeds W, additional write steps are invested. We have no idea how to analyze this algorithm. Therefore, in [5.35] a different algorithm, THROTTLE, is proposed that only admits $(1-\epsilon)D$ blocks per time step to the buffers. Then it is quite easy to show using queuing theory that the expected sum of the queue lengths is close to $D/(2\epsilon)$. Further, it can be shown that the sum of the queue lengths is concentrated around its mean with high probability so that a slightly larger buffer suffices to make waiting steps rare.[5]

Still, in many practical situations EAGER is not only simpler but also somewhat more efficient. Was the theoretical analysis futile and misguided? One of the reasons why we think the theory is useful is that it suggests a nice explanation of the measurements shown in Fig. 5.1. It looks like $1 - D/(2W)$

[5] The current proof shows that $W \in \mathcal{O}(D/\epsilon)$ suffices but we conjecture that this can be sharpened considerably using more detailed calculations.

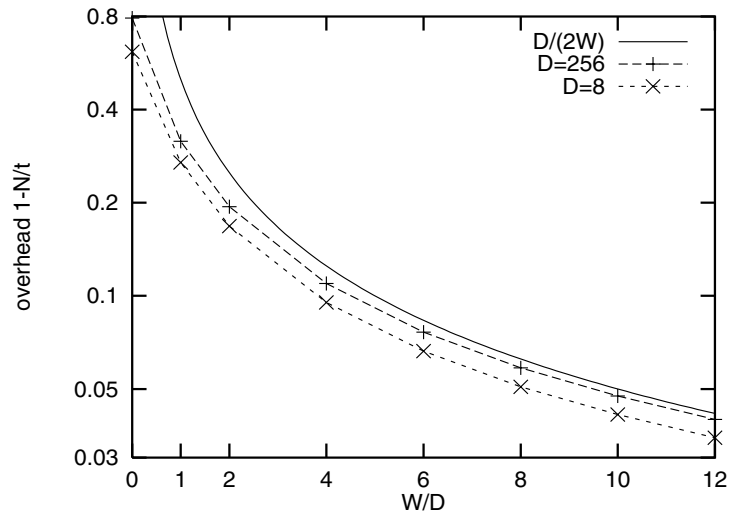


Fig. 5.1. Inefficiency (i.e., 1−efficiency) of EAGER. $N = 10^6 \cdot D$ blocks were written

is a lower bound for the average efficiency of EAGER and a quite tight one for large D. This curve was not found by fitting a curve but by the theoretical observation that algorithm THROTTLE with $\epsilon = D/(2W)$ would have buffer requirement about W.

More generally speaking, the algorithms we are most interested in might be too difficult to understand analytically. In such cases it makes sense to analyze a related and possibly inferior algorithm, and to use the scientific method to develop theoretical insights about the original algorithm. In the next Section we see that rather than simplifying the algorithm we can also simplify the analysis and achieve a similar effect — a theory in the sense of the scientific method.

5.3.2 "Heuristic" Deduction: Random Polling

Let us consider the following simplified model for the startup phase of *random polling dynamic load balancing* [5.21, 5.9, 5.34] which is perhaps the best available algorithm for parallelizing tree shaped computations of unknown structure: There are n processing elements (PEs) numbered 0 through $n-1$. At step $t = 0$, a random PE is busy while all other PEs are idle. In step t, a random shift $k \in \{1, \ldots, n-1\}$ is determined and the idle PE with number i asks PE $i + k \bmod n$ for work. Idle PEs which ask idle PEs remain idle; all others are busy now. How many steps T are needed until all PEs are busy? A trivial lower bound is $T \geq \log n$ steps since the number of busy PEs can

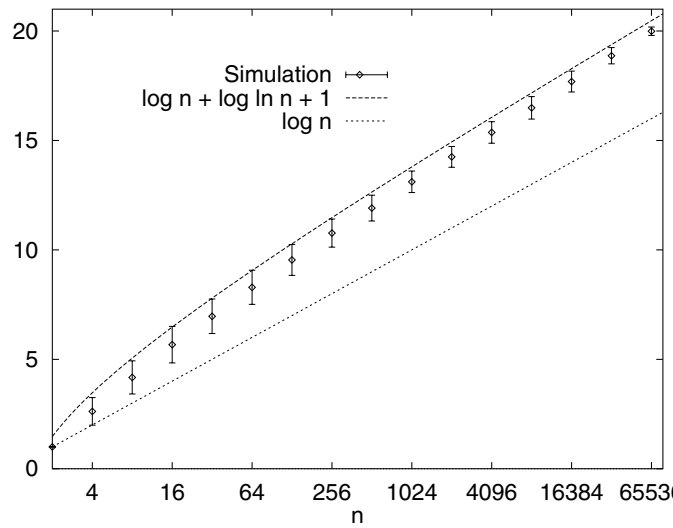


Fig. 5.2. Number of random polling steps to get all PEs busy: Hypothesized upper bound, lower bound and measured averages with standard deviation

at most double in each step. An analysis for a more general model yields an $E[T] \in \mathcal{O}(\log n)$ upper bound [5.34].

We will now argue that there is a much tighter upper bound of $E[T] \leq \log n + \log \ln n + 1$. We start with a theoretical analysis and get stuck half way. We then make a simplifying assumption (independence) that allows us to complete the analysis. The hypothesis generated in this way is then validated experimentally.

Define the 0/1-random variable X_{ik} to be 1 iff PE i is busy at the beginning of step k. For fixed k, these variables are identically distributed and $P[X_{i0} = 1] = 1 - 1/n$. Let $U_k = \sum_{i<n} X_{ik}$. We have

$$E(U_k) = E(\sum_{i<n} X_{ik}) = \sum_{i<n} P[X_{ik} = 1] = nP[X_{ik} = 1].$$

Since the X_{ik} are not independent even for fixed k, we are stuck with this line of reasoning. However, if we (falsely) assume independence, we get

$$P[X_{i,k+1} = 0] = P[X_{ik} = 0] \sum_{j \neq i} \frac{1}{n-1} P[X_{jk} = 0] = P[X_{ik} = 0]^2,$$

and, by induction,

$$P[X_{ik} = 0] = (1 - 1/n)^{2^k} \leq e^{-2^k/n}.$$

Therefore, $E(U_k) \geq n(1 - e^{-2^k/n})$ and for $k = \log n + \log \ln n$, $E(U_k) \geq n - 1$. One more step must get the last PE busy.

We have tested the hypothesis by simulating the process 1000 times for $n = 2^j$ and $j \in \{1, \ldots, 16\}$. Fig. 5.2 shows the results. On the other hand, the measurements do exceed $\log n + \log \ln n$. We conjecture that our results can be verified using a calculation which does not need the independence assumption.

5.3.3 Shellsort

Shellsort [5.39] is a classical sorting algorithm which has been widely studied. Given an increasing integer sequence of offsets h_i with $h_0 = 1$, the following pseudo-code describes Shellsort.

```
for each offset h_k in decreasing order do
    for j := h_k to n step h_k do
        x := data[j]
        i := j − h_k
        while i ≥ 0 ∧ x <data[i] do
            data[i + h_k] := data[i]
            i := i − h_k
        od
        data[i + h_k] := x
```

Despite its long history, Shellsort still poses several open problems. For example, let $T(n)$ denote the average number of key comparisons performed by Shellsort for n inputs. It is unknown whether there is an offset sequence which yields a sorting algorithm with $T(n) \in \mathcal{O}(n \log n)$ or even one with $T(n) \in o(n \log^2 n)$ [5.38, 5.18]. It is known that any algorithm with $T(n) = \mathcal{O}(n \log n)$ must use $\Theta(\log n)$ offsets [5.18]. Previous experiments with many carefully constructed offset sequences led to the conjecture that no sequence yields $T(n)$ close to $\mathcal{O}(n \log n)$ [5.45].

Motivated by the successful use of randomness for sorting networks [5.22, Section 3.5.4] where no comparably good deterministic alternatives are known, we asked ourselves whether *random* offsets might work well for Shellsort. For our experiments we used offsets which are the product of random numbers. The situation now is more difficult than in Section 5.3.2 where the theory gave us a very accurate hypothesis. Now we have little information about the dependence of the performance on n. Still, we should put the little things we do know into the measurements. First, by counting comparisons we can avoid the pitfalls of measuring execution time directly. Furthermore, we can divide these counts by the lower bound $\log(n!) \approx n \log n - n/\ln(2)$ for comparison based sorting algorithms. The difficult part is to find an adequate model for the resulting quotient plotted in Fig. 5.3. According to the conjecture in [5.45] the quotient should follow a power law. In a semilogarithmic plot this should be an exponentially growing curve. So this conjecture is not a good model at least for realistic n (also remember that Shellsort is usually

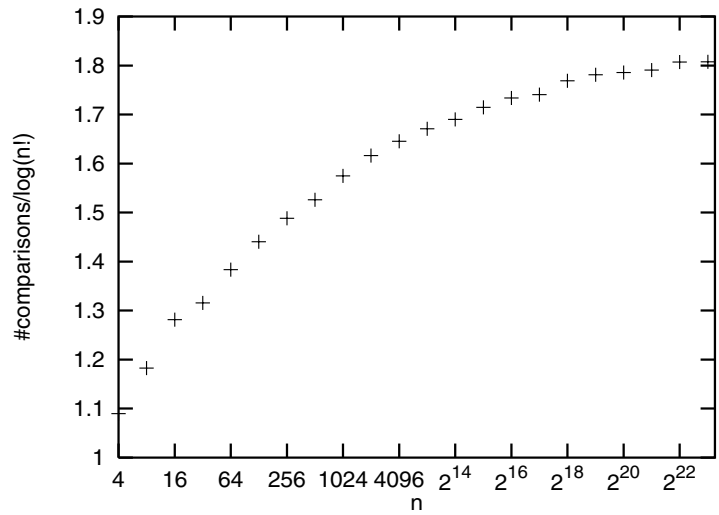


Fig. 5.3. Ratio of the average number of key comparisons of random offset Shellsort compared to the information theoretic lower bound $\log(n!)$. We used $h_i := \lfloor h_{i-1} \cdot f_i + 1 \rfloor$ where f_i is a random factor from the interval $[0, 4]$. Averages are based on 1000 repetitions for $n \leq 2^{13}$ and 100 repetitions for larger inputs

not used for large inputs). A sorting time of $\mathcal{O}(n \log^a n)$ for any constant $a > 1$ would result in a curve converging to a straight line in Fig. 5.3. Indeed, the curve gets flatter and flatter and its inclination might even converge to zero.

We might be tempted to conjecture that $T(n) = \mathcal{O}(n \log^{1+o(1)} n)$. But we must be careful here, because assertions like "$T(n) = O(f(n))$" or "the inclination of $g(n)$ converges to zero" are not experimentally falsifiable.

5.3.4 Sharpening a Theory: Randomized Balanced Allocation

Consider the following load balancing algorithm known as *random allocation*: m jobs are independently assigned to n processing elements (PEs) by choosing a target PE uniformly at random. Using Chernoff bounds, it can be seen that the maximum number of jobs assigned to any PE is

$$l_{\max} = m/n + \mathcal{O}(\sqrt{(m/n) \log n} + \log n)$$

with high probability (*whp*). For $m = n$,

$$l_{\max} = \Theta(\log(n)/\log\log n)$$

whp can be proven.

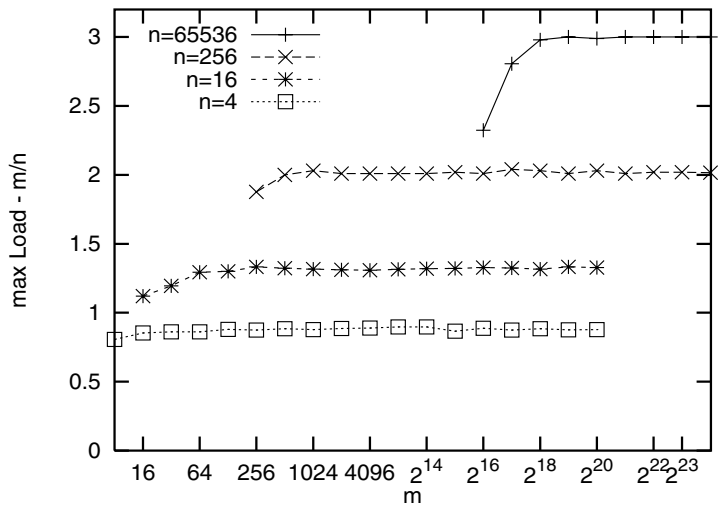


Fig. 5.4. Excess load for randomized balanced allocation as a function of n for different n. The experiments have been repeated at least sufficiently often to reduce the *standard error* $\sigma/\sqrt{\text{repetitions}}$ [5.32] below one percent of the average excess load. In order to minimize artifacts of the random number generator, we have used a generator with good reputation and very long period $(2^{19937}-1)$[5.24]. In addition, we have repeated some experiments with the Unix generator `srand48` leading to almost identical results

Now consider the slightly more adaptive approach called *balanced random allocation*. Jobs are considered one after the other. Two random possible target PEs are chosen for each job and the job is allocated on the PE with lower load. Azar et al. [5.2] have shown that

$$l_{\max} = \mathcal{O}(m/n) + (1 + o(1)) \log \ln n$$

whp for $m = n$. Interestingly, this bound shows that balanced random allocation is exponentially better than plain random allocation. However, for large m their methods of analysis yield even weaker bounds than that for plain random allocation. Fig. 5.4 shows that a simple experiment predicts that $l_{\max} - m/n$ cannot depend much on m. Recently[6] Berenbrink et al. [5.8] have published a proof (using quite nontrivial arguments) that indeed,

$$l_{\max} = m/n + (1 + o(1)) \log \ln n.$$

Our experiments were done before the theoretical solution. For other examples, we could have picked one of the other open problems in the area of balls into bins games. For example, Vöcking [5.43] recently proved that an

[6] After our experiments were done.

asymmetric placement rule for breaking ties can significantly reduce $l_{\max}$ for $m = n$ but nobody seems to know how to generalize this result for general m.

5.4 Empirical Curve Bounding Rules

We now develop several heuristic rules for finding asymptotic trends in data sets. To emphasize the general applicability of these techniques of data analysis, and to achieve some notational compatibility with related works in data analysis, we use the symbol x rather than n to refer to the parameter that is controlled during experimentation.

We begin with some notation and a precise specification of the problem. The cost of algorithm A is described by an unknown exact function $f(x)$, where x may denote problem size. An experiment produces a pair of vectors X, Y such that $Y[i] = F(X[i])$; in cases with randomized inputs and/or randomized algorithms, the experiment produces X, Y such that $E(Y[i]) = f(X[i])$ (that is, f is a function describing the average behavior of the algorithm). By convention, the vector X is assumed to contain k distinct nonnegative values arranged in increasing order.

The complexity class $\mathcal{O}(g(x))$ denotes a set of functions: we have $f(x) \in \mathcal{O}(g(x))$ if there exist positive constants c_u, x_u such that $0 \leq f(x) \leq c_u g(x)$ for all $x \geq x_u$. Similarly, $f(x)$ is in the set $\Omega(g(x))$ if there exist positive constants c_l, x_l such that $0 \leq c_l g(x) \leq f(x)$ for all $x \geq x_l$.

By convention, a complexity class is always labeled by the "simplest" member of the set; thus while $\mathcal{O}(3x^2 + 4x)$ is technically correct, we would use $\mathcal{O}(x^2)$ to denote this class. Throughout, $g(x)$ and $\bar{g}(x)$ are assumed to be simple functions labeling complexity classes, while $f(x)$ and $\bar{f}(x)$ may be arbitrary functions. The bar notation denotes functions that are estimates, and functions without bars denote (typically unknown) target functions.

Each heuristic rule takes X, Y, and reports a class estimator $\bar{g}(x)$ together with a bound type, either *upper*, *lower*, or *close*. *Upper* signifies a claim that $f(x) \in O(\bar{g}(x))$, and *lower* signifies a claim that $f(x) \in \Omega(\bar{g}(x))$. A rule will report a bound of *close* when the data is "too close to call" with respect to the upper/lower bound criteria being used. Usually this occurs when the data is indeed very close to the estimate, but in some cases a *close* result is returned because of some unexpected property of the data set.

An upper bound estimate $\mathcal{O}(\bar{g}(x))$ is *correct* if in fact $f(x) \in \mathcal{O}(g(x))$. A correct upper bound is *exact* if it labels the smallest correct class that holds the target function. Analogous definitions hold for lower bound estimates. Some heuristics iteratively generate internal *guess functions* $\bar{f}(x)$ stopping when come criteron is met and then reporting the corresponding estimate $\bar{g}(x)$ obtained from the leading term of $\bar{f}(x)$.

We consider the five strategies outlined below.

– The *Guess-Ratio* (GR) rule "guesses" a function $\bar{f}(x)$ and evaluates the guess according to the apparent convergence of the ratios $Y/\bar{f}(X)$.

- The *Guess-Difference* (GD) rule also guesses a function $\bar{f}(x)$, but evaluates the differences $\bar{f}(X) - Y$ rather than ratios.
- The *Power* (PW) rule combines log-log transformation of X and Y, linear regression on the transformed data, and residuals analysis. Two variations PW3 and PWD are introduced that improve this method for curve-bounding problems.
- The *Box Cox* (BC) rule combines a parametric transformation of Y values with linear regression and residuals analysis.
- The *Difference* (DF) rule generalizes Newton's divided difference method for polynomial interpolation. The generalization ensures that the method is defined and terminates for any data set.

Oracle Functions. In general, the rules can be viewed as interactive tools or as offline algorithms. To accommodate both views, we describe the algorithms in terms of a small set of *oracle functions* which decide, for example, whether "residuals are concave upwards." When the rules are used interactively, a human provides the oracle values; when the rules are offline, simple computations are used for each oracle function.

Trend(X, Y, c_r). Returns a value indicating whether Y appears to be *increasing* with X, *decreasing*, or *neither*. Our implementation compares the correlation coefficient r, computed on X and Y, to a cutoff parameter c_r which is 0.1 by default.

Concavity (X, Y, s). This function performs a linear regression on X and Y, smooths the residuals, and examines the signs of the smoothed residuals. It returns "concave upward" if signs obey the regular expression $(+)^+(-)^+(+)^+$ (at least one plus, followed by at least one minus, followed by at least one plus); it returns "concave downward" if they obey $(-)^+(+)^+(-)^+$; and otherwise the function returns "neither." The parameter s can be used to adjust the smoothing operation; the default low setting produces "less smooth" residuals and more frequent "neither" results.

DownUp(X, Y, s). The DownUp oracle examines smoothed Y values to determine whether Y appears to be first decreasing and then increasing within its range. If successive differences in smoothed Y values obey the regular expression $(-)^+(+)^+$, the function returns `True`; otherwise it returns `False`. The default low setting of parameter s (identical in purpose to the one for Concavity) produces less smooth values and more frequent `False` results.

NextCoef(f, *direction*, *cstep*) and NextOrder(f, *direction*, *estep*). Rules that iterate over several guesses require an oracle to supply the next guess. Our implementation constructs functions $f(x) = ax^b$ for positive rationals a and b. *NextCoef* changes a according to *direction* (up or down) and the *cstep* size. If a decrement of size *cstep* would give a negative coefficient, then *cstep* is reset to *cstep*/10 before decrementing. *NextOrder* changes the exponent b according to the *estep* size. In our tests the default *estep* is .001 for all but one rule, and the initial *cstep* value is .01.

The remainder of this section presents a "justification" for each rule in the form of a family of functions for which the rule is guaranteed to produce correct results.

5.4.1 Guess Ratio

To justify the Guess Ratio (GR) rule, let the set F_{GR} contain functions of the form $f(x) = a_1x^{b_1} + a_2x^{b_2} + \cdots + a_tx^{b_t}$, with rationals a_i positive, and rationals b_i such that $b_1 > 0$, $b_i \geq 0$, and $b_i > b_{i+1}$. Let the guess function be of the form $\bar{f}(x) = x^b$. Then the ratio $f(x)/\bar{f}(x)$ has the following properties: (1) When $f_1(x) \in O(\bar{f}(x))$, the ratio decreases to a nonnegative constant as x increases; (2) When $f_1(x) \notin O(\bar{f}(x))$ the ratio eventually increases and has a unique minimum point at some location x_r. If $x_r > 0$, then the ratio shows an initial decrease followed by an eventual increase. These properties are established by an application of Descartes' Rule of Signs [5.44] which (when extended from polynomials to functions in F_{GR} having rational exponents and coefficients) bounds the number of sign changes in the derivative of the ratio.

The Guess Ratio rule exploits this property by guessing a function $\bar{f}(x)$ and examining the ratio obtained for the finite sample X, Y. If a plot of X vs $Y/\bar{f}(X)$ shows an eventual increasing trend (perhaps with an initial decrease at low X values), then case (2) must hold. If only a decrease is observed in the plotted values, then cases (1) and (2) cannot be distinguished.

The Guess Ratio rule begins with a constant guess function $\bar{f}(x) = x^0$, and increments the exponent b using the NextOrder oracle, iterating until the ratios $Y/\bar{f}(X)$ do not appear to eventually increase. The Trend oracle is used to determine whether the ratios increase. The largest guess $\bar{f}'(x)$ for which an eventual increase is observed is reported as a "greatest lower bound" on the target $f(x)$: thus this rule always generates a *lower* claim that $f(x) = \Omega(\bar{g}_l(x))$, using the estimate $\bar{g}_l(x) = \bar{f}'(x)$.

When $f(x) \in F_{GR}$ and $k \geq 2$, the correctness of GR can be guaranteed simply by defining "eventual increase" as $Y[k-1] < Y[k]$ (recall that k is the size of X). However our implementation uses the Trend oracle (which calculates the correlation coefficient) for this test because of possible random noise in Y. Thus for any data set (X, Y) and for our Trend oracle, the rule must eventually terminate, but cannot be guaranteed correct.

5.4.2 Guess Difference

The Guess Difference (GD) rule also iterates over several guess functions $\bar{f}(x)$, evaluating differences $\bar{f}(X) - Y$ rather than ratios. It produces an upper rather than a lower bound estimate.

This rule is guaranteed correct for the set F_{GD} which contains functions $f(x) = cx^d + e$ where c, d and e are positive rationals, by the following

argument. Let the guess function have the form $\bar{f}(x) = ax^b$, and consider the *difference curve* $\bar{f}(x) - f(x)$. When $\bar{f}(x) \notin O(f(x))$, this curve must eventually increase (when x is "large enough"), and it must have a unique minimum at some location x_d. Also, note that x_d is inversely related to the coefficient a in the guess: for large a the difference curve increases everywhere ($x_d = 0$), but for small a there might be an initial decrease at small x. In the latter case we say the curve has the *DownUp* property.

The GD rule starts with an upper bound guess $\bar{f}(x) = ax^b$ and searches for a difference curve having the DownUp property by adjusting the coefficient a. If a DownUp curve is found, the rule concludes that $\bar{f}(x)$ overestimates the order of $f(x)$, so it decrements the exponent b and tries adjusting a again. The lowest b for which the rule finds a DownUp curve is reported as a "least upper bound" found. Thus if the rule stops at $\bar{f}'(x) = a'x^{b'}$, it reports an upper bound $f(x) = O(\bar{g}_u(x))$ with $\bar{g}_u(x) = x^{b'}$.

Using an analysis similar to that for GR, we can show that when $f(x) \in F_{GD}$ and X is fixed and when $k \geq 4$, then there exists an a such that $\bar{f}(X) - f(Y)$ will have the DownUp property. If the rule is able to find the a that produces a DownUp curve in its finite sample, then the upper bound it returns must be correct. In our implementation, if the rule is unable to find an initial DownUp curve within preset limits on iteration, the rule stops and reports the original guess provided by the user.

Note that Guess Difference rule cannot be guaranteed correct for functions from F_{GR} (defined for the Guess Ratio rule), because these functions may have several non-constant terms. If t is the number of terms in $f(x)$, and if $\bar{f}(x)$ over-estimates the order of $f(x)$, then the difference curve $\bar{f}(x) - f(x)$ can have at most $t - 1$ local minimal points (down-up-down-up-down-up) before its eventual increase. A DownUp curve in the plot for the finite sample may only be some initial fluctuation at small x, and it is not necessarily the case that $\bar{f}(x)$ overestimates $f(x)$.

5.4.3 The Power Rule

Power Rule (PW) modifies a standard data analysis technique for fitting curves to data. Suppose that the set F_P contains functions $f(x) = cx^d$ for positive rationals c and d. Let $y = f(x)$. Applying the logarithmic transformation $x' = \ln(x)$ and $y' = \ln(y)$, we obtain $y' = dx' + c$. Now y' is linear in x', and the slope obtained by a linear regression fit of x' to y' is equal to d, the exponent in the original function.

The Power Rule applies this log-log transformation to the data sets X and Y and then reports d, the slope of a linear regression fit on the transformed data. Since we are interested in bounds rather than fits, the Concavity oracle is applied to residuals from the linear regression fit. If the residuals appear to be concave upward, then the rule concludes that the data is growing faster than the fit, and returns a "lower" bound claim. If the residuals are concave downwards, the the rule returns "upper." If the residuals do not meet the

convexity criteria for these two claims, the oracle returns "neither" and the Power Rule returns "close."

If $Y = f(X)$ and $f(X) \in F_P$ then the Power rule finds the exponent d exactly. If Y is a random variate such that $Y = f(X) \cdot \epsilon$ and the random noise component ϵ obeys standard assumptions of independence and lognormality, then confidence intervals on the estimate of d can be derived by standard techniques (see [5.33] for details).

High-End Power Rule (PW3). When $f(x)$ contains low-order terms (such as $ax^b + e$), the log-log transformed points do not lie on a straight line. In this case, a linear regression using only the transformed points at the j highest X values might give a better asymptotic bound than one using all k points. The PW3 variation on the Power Rule applies the Power rule to the three highest data points corresponding to $X[k-2]$, $X[k-1]$, and $X[k]$.

Power Rule with Differences (PWD). The *differencing* variation on the Power rule attempts to straighten out plots under log-log transformation by removing constant terms. This variation can be applied when the X values are chosen such that $X[i] = \Delta \cdot X[i-1]$ for a positive constant Δ (for example, if $\Delta = 2$ then the X values are obtained by successive doubling. This variation applies the Power rule to *successive differences* in adjacent Y values, rather than to Y values alone.

To justify this rule, suppose F_{PWD} contains $f(x) = cx^d + e$ where c, d and e are positive rationals, and let $Y = f(X)$. Set $Y'[i] = Y[i+1] - Y[i]$ and $X'[1..k-1] = X[1..k-1]$.

Then we have

$$\begin{aligned} Y'[i] &= f(X[i+1]) - f(X[i]) \\ &= cX[i+1]^d + e - cX[i]^d - e \\ &= c(\Delta X[i])^d - cX[i]^d \\ &= c(\Delta)^d X[i]^d - cX[i]^d \\ &= X[i]^d(c\Delta^d - c) \end{aligned}$$

Now $Y' = c'X'^d$: that is, the exponent is the same as in the original, there is a new coefficient, and the constant e has been removed. The Power rule is then applied to Y' and X' in order to bound the exponent d. If $f(x) \in F_{\mathrm{PWD}}$, $Y = f(X)$ and $k > 4$, then the PWD rule is guaranteed to find d exactly.

Note that it is straightforward to extend this result to show that taking differences on Y twice will remove a logarithmic term.

5.4.4 The BoxCox Rule

To generalize the power rule, a standard approach in curve-fitting is to find transformations on Y or on X, or both, that produce a straight line in the transformed scale, and then to invert the transformation to obtain an estimate of the original curve. For example, if $Y = X^2$, then a plot of X vs $\sqrt{Y}$ would

produce a straight line, as would a plot of X^2 vs Y. One difficulty with the general approach is that it can be hard to find a good statistic to compare the quality of different transformations because the transformation changes the scale of the data points.

The Box-Cox ([5.1, 5.10]) curve-fitting method applies a transformation on Y that is parameterized by λ, and defines a "straightness" statistic that permits comparisons of transformations across different parameter levels. The transformation is as follows:

$$Y^{(\lambda)} = \begin{cases} \frac{Y^\lambda - 1}{\lambda Y^{\lambda-1}} & \text{if } \lambda \neq 0 \\ \\ \bar{Y} \ln(Y) & \text{if } \lambda = 0 \end{cases}$$

where $\bar{Y}$ is the geometric mean of Y, equal to exp(mean(ln (Y))). The "straightest" transformation in this family minimizes the Residual Sum of Squares (RSS) statistic which is calculated from X and Y^λ.

Our BC rule iterates over a range of guesses $\bar{f}(x) = x^b$ generated by the NextOrder oracle (with the range specified by the user). The rule evaluates $Y^{(\lambda)}$ with $\lambda = 1/b$ at each iteration, and the b' that produces the minimum RSS statistic is returned as the complexity class estimate $\bar{g}(x) = x^{b'}$. The Concavity oracle is then applied to residuals from the linear regression fit under the transformation, to determine the type of bound claimed (upper, lower, close).

When $f(x) = F_{PW}$, $Y = f(X)$, $k > 2$, and when NextGuess oracle includes $f(x)$, this rule is guaranteed to finds the function exactly. With standard normality assumptions about an additive random error term, it is possible to calculate confidence intervals for the estimate on exponent b: see [5.1] or [5.10] for details.

5.4.5 The Difference Rule

The **Difference** heuristic extends Newton's divided difference method for polynomial interpolation (see [5.40] for an introduction) This method calculates $Y^1 = \text{diff}(Y)/\text{diff}(X)$, where diff$(Y)$ denotes the differences between successive values in Y (and is therefore of length $k-1$), and $X^1 = X[1 \ldots k-1]$. If after d such calculations the resulting Y^d values are all equal, then we can conclude that $f(x)$ is a polynomial of degree d.

The extension used here applies when when Y contains random noise and nonpolynomial terms. The method iterates numerical differentiation on X and Y until the data "appears to be non-increasing," according to the Trend oracle. The number of iterations d required to obtain this condition provides an upper bound guess $\bar{g}(x) = x^d$. If $f(x)$ is a positive increasing polynomial of degree d, and if $k > d$, and $Y = f(X)$, then this method is guaranteed correct. Much is known about numerical robustness, best choice of design points, and (non)convergence when $k \leq d$.

5.4.6 Two Negative Results

A basic requirement is that a curve-bounding heuristic be internally consistent. For example, it should not be possible to reach the contradictory conclusions "Y is growing faster than X^2" and "Y is growing more slowly than X^2" on the same data set, merely by applying variations on the heuristic rule. Surprisingly, two plausible approaches included in our initial study turned out to have exactly this failure.

The first, perhaps the most obvious approach to the problem of bounding empirical curves, is to use general (nonlinear) regression to fit a multi-term function $\bar{f}(x)$ to the data set. The leading term of $\bar{f}(x)$ would provide the complexity class estimate, and the curvature of the residuals from regression analysis would provide the upper/lower bound claim.

Several general regression methods are known in the literature. These methods can be viewed as simple types of heuristic search, where a "step" from the current model $\bar{f}_i(x)$ to the next involves the addition or removal (or both) of an additive term, and the objective function (to be minimized) is a goodness-of-fit statistic such as the residual sum of squares (RSS).

In preliminary tests we found the RSS to be woefully inadequate for curve-bounding problems, in the sense that the statistic was quite oblivious to how close the leading term of $\bar{f}(x)$ was to that of true function $f(x)$. Nor were we able to discover a substitute statistic that could distinguish between a variety of guesses having different leading terms. As a result, when experimenting with this general regression method there was no sense of "convergence" towards a correct answer, and our "final" results were primarily artifacts of the stepping rule applied during the heuristic search. It seems an interesting problem for future research to determine whether general regression can be adapted to the curve-bounding problem.

The second approach is based on Tukey's [5.42] "ladder of transformation" technique, by which the X or Y values (or both), are transformed according to functions along the scale

$$\ldots x^{-1}, x^{-1/2}, \log(x), x^{1/2}, x^1, x^2 \ldots,$$

until the transformed data appears as a straight line. The best transformation on X, or inverse of the best transformation on Y, produces the asymptotic bound $g(x)$.

We implemented two versions of this approach, one which systematically applies transformations to Y, and one which transforms X. The straightness of each transformation was assed by the RSS statistic with respect to a linear regression on the transformed data; the upper/lower bound was determined by the Concavity oracle (or by visual inspection).

Our preliminary investigation showed that this approach frequently gives contradictory results depending on whether the transformation is applied to Y or X. The problem is that the correct transformation for the leading term of $f(X)$ can be difficult to find when a large (or even moderately-sized)

second-order term is present, and the importance of the second-order term varies considerably depending on whether Y or X is transformed. In our early tests these two rules frequently gave contradictory bound claims, such as both $\Omega(x^{2.2})$ and $O(x^{1.8})$.

As a result of these early failures, these two approaches were abandoned prior to the developement of the designed experiments, and are not considered further here. Note that the BoxCox curve-fitting method can be seen as a formalization of Tukey's transformation ladder (restricted to Y transformations), and some of the difficulties that we observe for BC may have similar foundation.

5.5 Experimental Results

The rules have been implemented in the S language [5.5], which is supported by the Splus software package designed for statistical and graphical computations. The main set of experiments were carried out on a Sun SPARCstation ELC, using functions running within Splus; some supporting experiments were conducted using the Lisp-based CLASP statistical/graphics package. Timing statistics would be very misleading in this context and are not reported in detail.

Roughly, however, the three Power rules required a few microseconds, and two of the iterative rules (Guess Ratio, BoxCox) usually took no more than a few seconds per trial (each trial corresponding to around 20-50 iterations of guess function generation). The Guess Difference rule iterates over two parameters (e and c), and was significantly slower than the other iterative rules; therefore a coarser *estep* value in the NextOrder oracle (0.01 instead of 0.001) was adopted to produce comparable wall clock times for this heuristic.

5.5.1 Parameterized Functions

The first experiment uses constructed functions $f(x) = ax^b + cx^d$, with $b > d$, with a positive, and with no randomization. To illustrate the sensitivity of the rules to low-order terms that may dominate at small x, this experiment varies the relative magnitudes of a to c and of b to d. Here the input vector X is small, containing powers of two ranging between 16 and 128.

Note that all of the successful examples in Section 5.3 use much larger problem sizes than are presented here. At any given maximum problem size, any curve-bounding rule will have no difficulty detecting asymptotic trends on "easy" functions having $b >> d$ and $a > c$. Similarly, any curve-bounding rule will fail on "hard" functions with $b \approx d$ and/or $a < c$. The goal of this experiment is to "stress" the rules and find the limits of successful applicability by using and difficult test functions for the given problem sizes.

To that end, the parameter values used in this experiment were selected (from the enormous space of possible combinations) after several weeks of

informal testing in order to locate the boundaries between easy and hard functions and problem sizes vfor these rules. Each parameter is allowed to vary within a range that causes some rules to move from success to failure. Curve-bounding rule that fail here will also tend to fail on harder functions and/or smaller problem sizes.

The exponent b takes three values $[0.2, 0.8, 1.2]$. Our initial exploration suggested that functions with exponents above two are generally quite easy to bound. Also, many open problems of interest to algorithm analyzers involve functions with exponents below two (see Section 5.3). Non-integer exponents were chosen here to avoid "lucky guesses" in our parallel tests using human oracles (since people tend to start guessing with integers). Similarly, the fixed coefficient $a = 3$ was chosen because people tend to guess one and ten first.

For each b value the second exponent d is set to $[0, 0.2, b - 0.2]$, subject to the restriction that $d < b$. The zero provides a constant second term, the 0.2 gives a second term which is "small" compared to b, and the third exponent is "near" b. For $d = 0$, the constant c is set to 10^4, and when $d > 0$ the coefficient c takes values from $[1, -1, 10^4]$ (small, negative, and large).

Figure 5.5 presents raw results from an experiment using all combinations of b, c, and d described above, plus three extra tests identified as functions 1, 2, and 11 (to illustrate some observations made below). In function 11 the constant 10^6 is added to ensure that all y values are positive, because some rules cannot handle negative y values.

The table shows the leading exponents that were returned by the rules. On functions 1 through 3, the correct exponent is 0.2; on functions 4 through 11 it is 0.8; and on functions 12 through 17 the exponent is 1.2. The notations (**l, u**) indicate the type of bound reported by the rule, either **lower** or **upper**. These numerical results have been rounded to two decimal places – lower bounds were rounded down, and upper bounds were rounded up. An underline marks a bound that is incorrect. A ***** marks a case where the heuristic failed to return an answer, usually because of lack of convergence.

Many intriguing observations arise.

The Guess Ratio (GR) rule, possibly the most widely-used curve-bounding technique in the folklore, performs surprisingly poorly. While it is frequently correct and close, it never dominates the three Power rules, and it always fails on functions having negative second terms (6, 9 and 16), even when the magnitude of the second term is small. This rule begins with a low guess function and iterates, increasing guesses, until the Trend oracle reports the ratio is "not increasing." With a negative second order term, the true function approaches its asymptote from above, which fools the oracle. A more sophisticated termination test might reduce this problem; but on the other hand we note in Section 5.6.1 that using a human to provide the termination test gives worse results in general.

Note that GR tends to "track" large positive second terms, producing correct, but less tight bounds, when the second term dominates the data.

	Function	GR	GD	PW	PW3	PWD	BC	DF
1	$3x^{0.2}+1$	0.17l	0.24u	0.17l	0.17l	0.20u	0.17l	1u
2	$3x^{0.2}+10^2$	0.01l	0.24u	0.01l	0.01l	0.20l	0.01l	1u
3	$3x^{0.2}+10^4$	0.00l	0.24u	0.00l	0.00l	0.20l	*	1u
4	$3x^{0.8}+10^4$	0.00l	*	0.00l	0.00l	0.80l	*	1u
5	$3x^{0.8}+x^{0.2}$	0.77l	*	0.77l	0.78l	0.79l	0.79l	1u
6	$3x^{0.8}-x^{0.2}$	0.82l	*	0.83u	0.82u	0.81u	0.81u	1u
7	$3x^{0.8}+10^4x^{0.2}$	0.20l	*	0.20l	0.20l	0.20l	0.20l	1u
8	$3x^{0.8}+x^{0.6}$	0.77l	*	0.77l	0.77l	0.77l	0.77l	1u
9	$3x^{0.8}-x^{0.6}$	0.83l	0.88u	0.85u	0.84u	0.83u	0.81l	1u
10	$3x^{0.8}+10^4x^{0.6}$	0.60l	*	0.60l	0.60l	0.60l	0.60l	1u
11	$3x^{0.8}-10^4x^{0.6}+10^6$	-0.01l	*	-0.06u	-0.09u	*	*	0u
12	$3x^{1.2}+10^4$	0.03l	1.3u	0.03l	0.05l	1.2l	*	2u
13	$3x^{1.2}+x^{0.2}$	1.18l	1.22u	1.18l	1.19l	1.19l	1.2u	2u
14	$3x^{1.2}+10^4x^{0.2}$	0.21l	*	0.21l	0.22l	0.26l	0.23l	1u
15	$3x^{1.2}+x^1$	1.17l	1.3u	1.17l	1.17l	1.17l	1.18u	2u
16	$3x^{1.2}-x^1$	1.23l	1.27u	1.25u	1.24u	1.24u	1.22l	2u
17	$3x^{1.2}+10^4x^1$	1.00l	*	1.00l	1.00l	1.00l	1.0l	1u

Fig. 5.5. Parameterized nonrandom functions. The numbers indicate the leading exponents returned by the rules. The notations **l, u**, indicate whether a **lower** or **upper** bound was returned. These numbers have been rounded to two decimal places – lower bounds were rounded down and upper bounds were rounded up. An underline marks a bound that is incorrect. The starred entries (*) mark cases where the rule failed to return a result

On functions 1, 2, and 3, for example, the bound actually decreases as the constant term becomes more important. Similarly, functions 3, 4, and 12 have the same constant second term, and in these three cases the bound returned by GR fails to follow the leading exponent. Finally, notice that performance deteriorates with respect to the function pairs (5 and 7), (13 and 14), and (15 and 17), which differ only in the coefficient on the second term.

The Guess Difference (GD) column contains several starred entries that mark cases where the rule failed to find an initial DownUp curve. In cases it returned the user-supplied starting guess, which was either $1x^1$ (functions 1 through 11) or $1x^2$ (functions 12 through 17). It appears that the performance of GD is quite sensitive to the choice of initial guess and step sizes: further exploration here suggests that the failures in functions 4 through 11, for example, are caused by an initial guess $1x^1$ that is too close to the true function $3x^{0.8}$. A higher initial guesses does allow the rule to get started and to find a tighter bound. Function 14 represents a different kind of failure – in this trial the GD routine was canceled after about 60 minutes of processing, at which time it was working on a guess of $1502.2x^{0.56}$, approaching the second order term from above.

However, when GD is able to get started, its estimates are surprisingly tight – much better than other rules in some cases. GD shows less sensitivity

to large second terms than does GR, but the rule is not impervious to second-order interference, as function 14 indicates.

The Power rules are close to one another, and also surprisingly close to GR in performance. However unlike GR, the three Power rules remain correct on functions 6, 9 and 16 (with negative second terms) by switching from "lower" to "upper" bound claims. Both PW3 and PWD give slightly tighter bounds than PW. Not only does PWD successfully eliminate the constant terms, producing exact bounds in functions 1–4 and 12, but it is slightly better than PW and PW3 even when the second term is not constant.

The BC rule returns bounds similar to those for GR and the Power rules. This rule provides very competitive bounds when it works, but it fails to converge on functions 3, 4, 11, and 12. These functions have a very large constant as a second term: it turns out that the failure of BC here is an intrinsic property of the λ transformation. That is, if the data is nearly constant, then the "straightest" transformation, having minimum RSS value, is obtained by the transformation $Y^{1/b}$ with $b = 0$. The rule iterates towards ever-smaller b values until the calculation of $1/b$ produces a numeric error.

Large increasing second terms (functions 7, 10, 14, 17) present no such termination problems for BC, although the rule does tends to track the second term. On functions 9, 15, and 16 the bound is incorrect although the estimate is close to those obtained by other rules. This appears to be due to interactions between the λ transformation and our Concavity function.

As is the case with PWD, the differencing operation performed by the DF rule eliminates the effect of large constant terms. Recall that this rule can only return integer exponents, which are often correct but rarely close to the selected functions. This rule fails on functions 11, 14, and 17.

Function 11 is disasterous for all the rules because the large negative second term causes Y to be decreasing within its range. As a general rule, these rules do not work well on functions that are decreasing or even temporarily decreasing within their range.

Increasing the Largest Problem Size. The obvious remedy to the problem of a dominant second-order term is to use larger problem sizes. The second experiment uses functions identical to those of the previous section, but X takes values at powers of two in the range $8 \ldots 256$ rather than $8 \ldots 128$ thereby doubling the largest problem size.

The results in Figure 5.6 are very similar to those in in the previous chart, suggesting that in general the rules respond very slowly to changes in the largest input values. In particular, doubling the largest problem size has very little effect on the bounds returned by Guess Ratio and the three Power Rules. The observed changes in estimates were generally only in the third or higher decimal places, and incorrect bounds remain incorrect.

The Guess Ratio rule could be made more responsive to changes in problem size if a different Trend oracle were used to provide the stopping condition: instead of calculating the correlation coefficient, an oracle that concen-

	Function	GR	GD	PW	PW3	PWD	BC	DF
1	$3x^{0.2}+1$	0.17l	0.23u	0.17l	0.17l	0.20u	0.18l	1u
2	$3x^{0.2}+10^2$	0.01l	0.23u	0.01l	0.01l	0.20l	0.01l	1u
3	$3x^{0.2}+10^4$	0.00l	0.23u	0.00l	0.00l	0.20l	*	1u
4	$3x^{0.8}+10^4$	0.00l	0.83u	0.00l	0.01l	0.80l	0.00l	1u
5	$3x^{0.8}+x^{0.2}$	0.77l	0.82u	0.77l	0.78l	0.79l	0.79l	1u
6	$3x^{0.8}-x^{0.2}$	0.82l	0.83u	0.83u	0.82u	0.81u	0.81u	1u
7	$3x^{0.8}+10^4x^{0.2}$	0.20l	*	0.20l	0.20l	0.20l	0.20l	1u
8	$3x^{0.8}+x^{0.6}$	0.77l	0.80u	0.77l	0.77l	0.77l	0.78c	1u
9	$3x^{0.8}-x^{0.6}$	0.83l	0.85u	0.84u	0.83u	0.83u	0.82c	1u
10	$3x^{0.8}+10^4x^{0.6}$	.60l	*	0.60l	0.60l	0.60l	0.60l	1u
11	$3x^{0.8}-10^4x^{0.6}+10^6$	-0.01l	*	-0.07u	-0.15u	*	*	0u
12	$3x^{1.2}+10^4$	0.06l	1.22u	0.05l	0.11l	1.20l	*	2u
13	$3x^{1.2}+x^{0.2}$	1.19l	1.22u	1.18l	1.19l	1.19l	1.20u	2u
14	$3x^{1.2}+10^4x^{0.2}$	0.22l	*	0.21l	0.23l	0.29l	0.25l	1u
15	$3x^{1.2}+x^{0.8}$	1.17l	1.20u	1.17l	1.18l	1.18l	1.19u	2u
16	$3x^{1.2}-x^{0.8}$	1.22l	1.24u	1.24u	1.23u	1.23u	1.21l	2u
17	$3x^{1.2}+10^4x^{0.8}$	0.80l	*	0.80l	0.80l	0.80l	0.80c	1u

Fig. 5.6. Doubling the largest problem size. The numerical values show the leading exponent returned by the rule. The notations **l, u, c**, indicate the type of bound reported by the rule, either **lower**, **upper**, or **close**. These results are rounded to two decimal places: lower bounds are rounded down, upper bounds are rounded up and close bounds are rounded to the nearest decimal. An underline marks a bound that is incorrect. A * marks a rule that failed to return an answer

trates on the high end of the data set might be more successful here. It is surprising that PW3 does not respond much to the change in problem size, because only the highest three data points are checked each time. One would expect the new point to have much greater leverage for this rule.

The greatest improvement is found in the Guess Difference (GD) rule on functions 4 through 9 (excepting 7). In the previous experiment the rule failed to find an initial DownUp curve at all—now the rule is able to find an initial curve, and iterate to find upper bounds within 0.05 of the true exponent. The BC rule also shows some very slight improvement: in two cases the rule produces *close* bound claims where previously the claim had been incorrect.

It is a problem for future research to how best to design rules that respond to significant changes in problem sizes. For now, it remains important in any algorithmic experiment to obtain results using the largest problem sizes possible, especially when the underlying function has low exponents.

Adding Random Noise. The previous two experiments use functions with no random noise in the data. In the third experiment we add a random term to three functions (1, 5, and 13) that were easy for all rules, to learn how rule performance degrades with increased variance. We let $Y = \bar{f}(X) + \epsilon_i$ with $i = 1, 2, 3$. The random variates ϵ_i are drawn independently from a normal distribution with mean 0 and standard deviation set to constants 1

($i = 1$) and 10 ($i = 2$), and to the function means $\bar{f}(X[j])$ ($i = 3$). We ran two independent trials for each i, in order to check for spurious positive and negative results. A table of results appears in Figure 5.7.

Not surprisingly, the quality of results returned by all rules degrades as dramatically as random variation increases. The replication of tests in each category demonstrates that many correct bounds are in fact spurious. Conversely, of course, rule performance improves when variance in the data decrease: This is good news for experimentors because is often possible to reduce variance in experimental data, either by increasing the number of trials or by applying one of several variance reduction techniques known in the literature (see [5.25]). Note that variance is less of a problem when the first term exponent is large enough.

The GR rule responds strangely to random data, returning negative bounds and lower bounds of 2.98 and even 25.7 [sic] on these functions. Not surprisingly, PW3 is frequently wrong – when random variation is present, it seems wise to make use of all the data, rather than just part of it. As

Function	GR	GD	PW	PW3	PWD	BC	DF
$3x^{0.2} + 1$	0.173l	0.23u	0.17l	0.17l	0.2c	0.18l	1u
$3x^{0.2} + 1 + \epsilon_1$	0.12l	*	0.15c	-0.00u	0.05u	0.90u	1u
$3x^{0.2} + 1 + \epsilon_1$	0.10l	*	0.10c	0.34u	-0.02l	0.40u	1u
$3x^{0.2} + 1 + \epsilon_2$	25.7l	0.57u	0.97u	0.67u	-0.5c	*	1u
$3x^{0.2} + 1 + \epsilon_2$	0.90l	*	0.63c	0.40l	0.19l	*	2u
$3x^{0.2} + 1 + \epsilon_3$	-0.1l	*	-0.01c	-0.55u	0.93l	0.41c	0u
$3x^{0.2} + 1 + \epsilon_3$	-0.01l	*	-0.05c	-0.34l	0.03c	1.00c	0u
$3x^{0.8} + x^{0.2}$	0.77l	0.82u	0.77l	0.78l	0.79l	0.79l	1u
$3x^{0.8} + x^{0.2} + \epsilon_1$	0.77l	0.83u	0.77l	0.77l	0.80u	0.78c	1u
$3x^{0.8} + x^{0.2} + \epsilon_1$	0.76l	0.78u	0.76c	0.81u	0.77l	0.81c	1u
$3x^{0.8} + x^{0.2} + \epsilon_2$	0.71l	*	0.75c	0.77u	0.78c	0.69c	1u
$3x^{0.8} + x^{0.2} + \epsilon_2$	0.69l	*	0.68c	0.73l	0.89c	0.81c	1u
$3x^{0.8} + x^{0.2} + \epsilon_3$	1.50l	*	1.34c	1.03u	0.91u	*	2u
$3x^{0.8} + x^{0.2} + \epsilon_3$	1.08l	*	1.01u	-0.35u	1.98u	*	1u
$3x^{1.2} + x^{0.2}$	1.19l	1.22u	1.18l	1.19l	1.19l	1.20u	2u
$3x^{1.2} + x^{0.2} + \epsilon_1$	1.18l	1.22u	1.18l	1.19l	1.21u	1.20c	2u
$3x^{1.2} + x^{0.2} + \epsilon_1$	1.18l	1.22u	1.18l	1.19l	1.19l	1.20c	2u
$3x^{1.2} + x^{0.2} + \epsilon_2$	1.18l	1.22u	1.17l	1.20u†	1.19u	1.19c	2u
$3x^{1.2} + x^{0.2} + \epsilon_2$	1.15l	1.30u	1.14l	1.18l	1.22c	1.22c	2u
$3x^{1.2} + x^{0.2} + \epsilon_3$	0.10l	1.99u	1.25l	2.20l	1.83l	*	1u
$3x^{1.2} + x^{0.2} + \epsilon_3$	2.98l	2.00u	1.58u	0.39u	0.94l	2.59u	1u

Fig. 5.7. Adding random noise. The numbers show the exponents returned by the rules. The notations **l, u, c**, indicate the type of bound reported by the rule, either **lower**, **upper**, or **close**. These results are shown rounded to two decimal places: lower bounds are rounded down, upper bounds are rounded up, and close bounds are rounded to the nearest decimal. The † marks a case where rounding changed an originally incorrect upper bound (1.194u) to a correct one (1.2u). An underline marks a bound that is incorrect. The starred entries (*) mark cases where the rule failed to return a bound

variance in Y increases, the Power and the BoxCox rules more frequently return claims of *close*. We do not know how to interpret these results to obtain bounds (upper or lower) on function growth; therefore these rules may be less useful for curve-bounding problems when large variance is present.

5.5.2 Algorithmic Data Sets

The experiment in this section applies the rules to eight data sets taken from previous computational experiments by the first author. The data sets were originally developed in the context of experimental research on algorithms, and not for testing curve-bounding heuristics. Thus the performance of the heuristics on these data sets may give more realistic indications of their performance in practice. On the other hand, since these data sets are from research problems, we don't always know the true underlying function $\bar{f}(x)$, and can't always tell when the rules are correct.

The results appear in Figure 1.8. The left column gives the best analytical bounds known for each function. The entries NA for PWD mark cases where this rule was not applied because design points were not in required format (with X increasing by constant multiples).

Data sets 1 and 2 represent the expected costs of Quicksort and Insertion Sort, formulas for which are known exactly (see for example [5.20]). The X values are [10, 20, 30, ..., 1000] for Quicksort, and [10, 20, 30 ..., 500] for Insertion sort. These data sets were generated from the formulas with no random noise. An experimental study of these algorithms would produce random variation in the data, but because these algorithms are extremely efficient it would be possible to make the variace quite small by taking large batches of trials. For Quicksort the asymptotic leading term (i.e. the "correct answer" is $\Theta(x \log x)$; for Insertion sort the leading term is $\Theta(x^2)$.

Sets 3 through 6 are from experiments on heuristics for one-dimensional bin packing [5.6], [5.7]. In these experiments X takes values [200, 400, 800, ..., 128000] (doubling each time). Set 3 shows measurements of *bin count* and Set 4 measures *empty space*, for First Fit Decreasing rule. Sets 5 and 6 show measurements of empty space for the First Fit rule under two different parameter settings. In all four cases, each Y value represents the mean of 25 independent trials. Variance in the four data sets is, respectively, about 0.3x, 40x, 1x ,0.1x (times) the mean. The formulas shown on the left represent the best analytical bounds known for the functions generating these data.

Sets 7 and 8 are from experiments on distances in random complete graphs having weights drawn from a uniform distribution on $(0, 1]$ [5.26]. In both cases X = [200, 400, 600, ..., 1400] and each Y value represents the mean of 50 independent trials. In Set 7 variance is about 2x mean, and in Set 8 variance is a constant near 1000.

Contrary to experience with the constructed functions, the Guess Ratio rule (GR) obtains a correct and tight bound when a negated second term

	Known	GR	GD	PW	PW3	PWD	BC	DF
1	$(x+1)(2H_{x+1}-2)$	1.20l	1.24u	1.23u	1.19u	NA	1.18c	2u
2	$(x^2-x)/4$	2.00l	2.03u	3.01u	3.01u	NA	2.00l	2u
3	$x/2+O(1/x^2)$	0.99l	*	0.99l	1.00u†	1.00c	1.20c	2u
4	$\Theta(x^{0.5})$	0.52l	*	0.55c	0.58u	0.78c	1.00c	1u
5	$O(x^{2/3}(\log x)^{1/2})$, $\Omega(x^{2/3})$	0.68l	0.72u	0.69c	0.69u	0.69c	0.69c	1u
6	$y \leq 0.68x$	0.90l	1.00u	0.89l	0.95l	1.26l	0.98c	1u
7	$x-1 \leq y \leq 13.5x \ln x$	1.13l	1.18u	1.15u	1.12l	NA	1.11c	2u
8	$x \ln x < y < 1.2x^2$	1.30l	1.47u	1.32u	1.20l	NA	1.20c	2u

Fig. 5.8. Data from algorithmic experiments. The numbers give the leading exponents returned by the rules. The notations **l, u, c**, indicate the type of bound reported, either **lower**, **upper**, or **close**. The numbers are rounded to two decimal places: lower bounds are rounded down, upper bounds are rounded up, and close bounds are rounded to the nearest decimal. The † marks a case where rounding changed an incorrect result (0.999u) to a correct one (1.00u). An underline marks a bound which is known to be incorrect, and * marks a case where the rule failed to return an answer. In some cases (NA) the PWD rule was not applied because the X values in the data did not increase by constant factors

is present (Set 2). However in four cases (Sets 1, 4, 5, and 7), GR produces lower bound claims that violate the known bounds.

For Set 1 (and possibly for Sets 5, 7, and 8), the leading term contains a logarithmic factor, which is not generated by our NextOrder function. From additional tests that include logarithmic terms as guess functions, we observe that none of the rules is able to distinguish logarithms from low-order exponents such as $x^{0.2}$ with any degree of reliability. Since logarithms do tend to occur in many algorithmic research problems, it would be useful to develop some techniques that can be applied specifically to this problem.

The Guess Difference rule and the Power Rules rarely violate known bounds on the data sets, although without better analyses it is impossible to tell whether the rules are correct in all cases. Note that BC nearly always returns a "close" report, which is very difficult to evaluate. Interestingly, every incorrect bound produced by these rules is a lower bound.

Data Sets 5 through 8 have gaps between the known lower and upper bounds. In these cases we might hope that the heuristic rules can provide some insight to direct future analytical research: does the upper bound need to be lowered, or does the lower bound need to be raised (or both)?

In Sets 5 and 7, the $(\log x)^{0.5}$ and $c \log x$ gaps are too small to be distinguishable by these rules. In Set 6, however, the rules provide consensus support for a conjecture that the true function $\bar{f}(x)$ is closer to linear $\Theta(x^1)$ than, say, to a square-root function $\Theta(x^{0.5})$. In Set 8 the results are even stronger. Given the above observation that logarithmic terms tend to be indistinguishable from terms near $x^{0.2}$, we have much greather support for a conjecture $\bar{f}(x) = \Theta(x \log x)$ than than $\bar{f}(x) = \Theta(x^2)$ although the true an-

swer may be somewhere in between. (In this case there is external supporting evidence that the lower bound is tight.)

5.6 A Hybrid Iterative Refinement Method

In our informal explorations and designed experiments with little or no random noise in the data, all the rules generally can get within a linear or sometimes $\sqrt{x}$ factor of the exact bound, except when they become "fooled" by very large second-order terms. It is possible to reduce the effect of large second-order terms by taking larger problem sizes, but the rules are surprisingly slow to respond to this type of change. In this section we describe a hybrid rule which appears to be very robust with respect to large second terms.

The hybrid rule incorporates an iterative diagnosis and repair technique that combines the existing heuristics to produce improved guess function modes. The technique is designed to find upper bounds for functions of the form $ax^b + cx^d$ with rational exponents $b > d \geq 0$ and real coefficients $a \ll c$. This method represents a departure from our approach up to now: The earlier methods were intended to be general, but this one is specific to functions with relatively large coefficients on low order terms. This suggests a new role for the methods we have discussed so far: Instead of using them to guess at the order of a function, they can provide diagnostic information about the function (e.g., whether $a \ll c$), and then more specific, purpose-built methods, designed for particular kinds of functions, can estimate parameters.

To illustrate this new approach, we developed a three-step hybrid method for functions of the form $\bar{f}(x) = ax^b + cx^d$;

1. Apply a discrete derivative (the Difference rule) to the datasets, in order to find the integer interval of the exponent b.
2. Refine the guess for the exponent using the Guess Ratio rule. We start with the known upper and lower bound for the exponent, u and l. At each step we consider the model $x^{(u+l)/2}$ by plotting x against $y/x^{(u+l)/2}$. If the plotted points appear to be decreasing, then $(u+l)/2$ is overestimating the exponent, and we replace u by $(u+l)/2$. If the points are increasing, then l will be replaced. The estimates are refined until u and l get within a desired distance ϵ of each other. At this point, if the dataset $y/x^{(u+l)/2}$ has a DownUp feature, then we know that function $\bar{f}$ must have a relatively high coefficient c on a low order term. This diagnosis invokes the next step.
3. If, as we suspect, the current result is tracking a low-order term with a high coefficient, then this term will dominate $\bar{f}$ for small values of x. Thus we can approximate the upper bound for small x's to be cx^d. Let (x_1, y_1) and (x_2, y_2) be two points from the beginning part of the curve. If we consider that $y_1 \approx cx_1^d$ and $y_2 \approx cx_2^d$, then d can be approximated

by $\frac{\log y_1 - \log y_2}{\log x_1 - \log x_2}$, and c is $\frac{y_1}{x_1^d}$. Now we can correct the model using these estimates, in order to make the high-order term appear. For all points (x, y), we transform y into $\frac{y}{x^d} - c$. Now we can apply the same procedure as above to find the a and b parameters, assuming that $y \approx ax^b$. In this case, though, we use for our estimates two points that have high values of x, as the influence of the high-order term is stronger for these points.

This technique illustrates a way in which models can be improved by generating data and comparing it against the real values to obtain diagnostic information (step 2), which suggests a method specific to the diagnosis—in this case, a method specific to functions with large coefficients on low order terms. (We envision similar diagnostics and methods for functions with negative coefficients, but we haven't designed them, yet.)

The results of this method are found in the columns labeled HY in Figures 1 and 2. The results are tight upper bounds when $\bar{f}$ does in fact contain a low order term with a large coefficient (functions 7, 10, 11, 14, and 17 in Figure 5.5). In fact, these bounds are tighter than those returned by the other methods, and, remarkably, this hybrid method estimates coefficients and low order exponents very well. When the functions do not contain low order terms with large coefficients, the bounds returned by this method remain correct but they are looser than those given by other methods. Interestingly, this situation is often indicated by very low estimated coefficients on the high order terms; for example, in funtion 1 (Fig. 1), the coefficient of the first term is 0.03. The only cases when the technique fails are those in which negative coefficients appear in the low-order terms. The failure is probably due to the sensitivity of the Guess Ratio heuristic to such circumstances. This new method was also tested on noisy datasets but the noise had negligible effects. The new method used different oracles and different implementations of oracles from the previous methods, which might account for the relatively robust performance. Or, the small effects of noise might be due to a different method for sampling data from the given functions. Clearly, the effects of noise on these methods are still poorly understood.

5.6.1 Remark

In our informal and designed experiments with little or no random noise in the data, all the rules generally can get within about a $\sqrt{x}$ factor of the exact bound, except when they become "fooled" by large or negative-valued second-order terms. It is possible to reduce the effect of large second-order terms by taking larger problem sizes, but the rules are slow to respond to this type of change. The hybrid diagnostic method described in Section 5.6 can be used with success on such problems.

On data from algorithmic research problems, the rules can return results within a factor of x and sometimes less (of the correct answer when it is known, and of one another when it is not known). The rules are not reliable

in distinguishing low-order and logarithmic factors (this holds even when logarithms are added to the NextOrder oracle). Thus while the simple rules applied here provide fairly reliable conjectures to guide future analytical research when the known bounds are separated by at least a linear factor, more sophisticated approaches (or perhaps better data sets) are necessary if finer distinctions are needed.

It is sometimes possible to improve the data sets to obtain more reliable results. Although the rules do not much respond when the largest problem size is doubled, they do seem to be very responsive to reductions in data variance. This is good news for algorithm analyzers, since variance can be reduced by taking more random trials, and trials are easier to get when Y grows slowly: the situations where small variance is most needed are those situations where small variance is easiest to obtain.

Can Humans Do Better? We have preliminary results concerning interactive uses of the rules. In one experiment, the fourth co-author was given the 25 data sets presented here, without any information about their provenance, and was allowed to use any data analysis approach available in the powerful CLASP library. The human was more frequently incorrect than any of the implemented rules, and the human/machine interactions took much more time to accomplish.

A second experiment involved strict application of the heuristic rules, but with a human oracle (the first co-author) who was familiar with the eight algorithmic data sets. Here also, interactive trials required much more time to perform than did the offline versions (on the order of a few hours rather than a few seconds). Very preliminary results indicate that: the GR produces worse (less close) bounds with a human Trend oracle; the human Concavity oracle tends to agree with the implemented one when used by the Power rules (no change in performance); a human-interactive version of the GD rule is more successful at finding initial DownUp curves (leading to more frequent success), but is not able to find tighter bounds for this rule in general; and an interactive BoxCox can be used to provide upper/lower bounds that bracket the estimate, thus avoiding the "close" and errorneous bounds returned by the implemented version.

Removing Constant Terms. In many applications it may be possible to remove a constant from Y before analysis, either by testing with $x = 0$ or by subtracting an estimated constant. Our preliminary results suggest that subtraction of a known constant uniformly improves all the rules, but subtracting an estimated constant gives mixed results.

Rule Variations. It is a problem for future research to implement and evaluate the many variations on the oracles and the iterative rules GR, GD, and BC. The Guess Ratio rule would probably be improved by a Trend oracle that is robust with respect to negated second terms. Indeed, it is likely that much more sophisticated oracle functions than our simple ones can be developed.

The Guess Difference rule appears to be very sensitive to the initial function and to the granularity of the step functions in the NextOrder and NextCoefficient oracles. So far we cannot find any pattern for this sensitivity. It does seem clear that when an initial guess is too close to the answer, GD fails to find an initial DownUp curve. This rule might be greatly improved by addition of a heuristic search mechanism. Also, we might give the iterative rules fewer options to choose from. The BoxCox rule sometimes improves with coarser step size (because the best transformation gives an exponent somewhere the first and second terms). When the fit is close, however, the BC rule can make erronous bound claims. Thus the rule's goal of finding the best fit works at odds with the goal of finding a reliable bound. The bounds returned by GR and GD nearly always improve when step size decreases. The PWD might be improved by taking differences more than once; one promising idea is to take differences until the data appears concave downwards.

5.7 Discussion

We have seen different aspect of the problem how to identify asymptotic behaviour from experiments. Sections 5.4–5.6 provide us with a few rather general semi-automatic tools for this purpose but also with plenty of examples where these rule do not work.

More successful is the more specific approach based on the scientific method discussed in Section 5.3. But in what sense are these examples "successful"? Assume that using the scientific method we have found an experimentally well supported hypothesis about the running time of an important, difficult to analyze algorithm. How should this result be interpreted? It may be viewed as a conjecture for guiding further theoretical research for a mathematical proof. If this proof is not found, a well tested hypothesis may also serve as a surrogate. For example, in algorithmics the hypotheses "a good implementation of the simplex method runs in polynomial time" or "NP-complete problems are hard to solve in the worst case" play an important role. The success of the scientific method in the natural sciences — even where deductive results would be possible in principle — is a further hint that such hypotheses may play an increasingly important role in algorithmics. For example, Cohen-Tannoudji et al. [5.13] (after 1095 pages of deductive results) state that "in all fields of physics, there are very few problems which can be treated completely analytically." Even a simple two-body system like the hydrogen atom cannot be handled analytically without making simplifying assumptions (like handling the proton classically). For the same reason, experiments are of utmost importance in chemistry although there is little doubt that well known laws like the Schrödinger equation in principle could explain most of chemistry.

Of course, no tool is perfect, and the hazards of extrapolating from experimental data to find reliable asymptotic bounds can not be ignored. Our

study of five simple heuristic strategies (with variations) suggests that any of the approaches can produce a correct asymptotic bound within an order of magnitude when the data set is well-behaved: that is, when there is very little random noise in the y-values, and when the largest problem size is large enough to overcome "noise" due to large constant factors in low order terms.

However, when the research problem requires inferences about bounds that are more finely-tuned than one order of magnitude (for example, whether a function grows as $O(n)$ vs $O(n \log n)$, or whether a root-n factor is present), the five rules become unreliable, especially when the quality of data deteriorates. The rules are quite sensitive to random variation in the y-values, and somewhat less sensitive to changes in the largest problem size.

In these types of experimental situations, then, the extrapolation techniques described here must be used with caution, and/or steps must be taken to improve the quality of the data obtained from the experiment. Fortunately, in many algorithmic research problems it is easy to reduce variance in the experimental data by taking more experiments or by applying variance reduction techniques. It does appear to be an important component of good experimental practice to set problem sizes as large as possible, so as to overcome any possible interference from low order terms.

It is an interesting open research problem to develop better and more sophisticated strategies for obtaining reliable asymptotic inferences from algorithmic experiments.

References

5.1 A. C. Atkinson. *Plots, Transformations and Regression: an Introduction to Graphical Methods of Diagnostic Regression Analysis*. Oxford University Press, U.K., 1987.

5.2 Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the 26th ACM Symposium on the Theory of Computation (STOC'94)*, pages 593–602, 1994.

5.3 D. Baldwin and J. A. G. M. Koomen. Using scientific experiments in early computer science laboratories. *ACM SIGCSE Bulletin*, 24(1):102–106, 1992.

5.4 R. S. Barr, R. V. Helgaon, and J. L. Kennington. Minimal spanning trees: An empirical investigation of parallel algorithms. *Parallel Computing*, 12:45–52, 1989.

5.5 R. A. Becker, J. A. Chambers, and A. R. Wilks. *The New S Language: A Programming Enviornment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole, 1988.

5.6 J. L. Bentley, D. S. Johnson, F. T. Leighton, and C. C. McGeoch. An experimental study of bin packing. In *Proceedings of the 21th Annual Allerton Conference on Communication, Control, and Computing*, pages 51–60, 1983.

5.7 J. L. Bentley, D. S. Johnson, C. C. McGeoch, and L. A. McGeoch. Some unexpected expected behavior results for bin packing. In *Proceedings of the 16th ACM Symposium on Theory of Computation (STOC'84)*, pages 279–298, 1984.

5.8 P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *Proceedings of the 32nd ACM Symposium on the Theory of Computation (STOC'00)*, 2000.
5.9 R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Symposium on Foundations of Computer Science (FOCS'94)*, pages 356–368, 1994.
5.10 G. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley & Sons, Inc., Chichester, 1978.
5.11 J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*. Duxbury Pres, Boston, 1983.
5.12 P. R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, Cambridge, MA, and London, England, 1995.
5.13 C. Cohen-Tannoudji, B. Diu, and F. Laloë. *Quantum Mechanics*, volume 2. John Wiley & Sons, Inc., Chichester, 1977.
5.14 P. J. Denning. What is experimental computer science? *Communications of the ACM*, 23(10):543–544, 1980.
5.15 P. J. Denning. Performance analysis: Experimental computer science at its best. *Communications of the ACM*, 24(11):725–727, 1981.
5.16 N. Fenton, S. L. Pfleger, and R. L. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(4):86–95, 1994.
5.17 J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.
5.18 T. Jiang, M. Li, and P. Vitányi. Average-case complexity of Shellsort. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*. Springer Lecture Notes in Computer Science 1644, pages 453–462, 1999.
5.19 D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms, 1996. Manuscript.
5.20 D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 2nd edition, 1998.
5.21 V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
5.22 T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.
5.23 P. Lukowicz, E. A. Heinz, L. Prechelt, and W. F. Tichy. Experimental evaluation in computer science: A quantitative case study. *Journal of Systems and Software*, 28(1):9–18, 1995.
5.24 M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.
5.25 C. C. McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 245(2):195–212, 1992.
5.26 C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13:426–441, 1995.
5.27 C. C. McGeoch. Toward an experimental method for algorithm simulation, 1996.
5.28 C. C. McGeoch and B. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.
5.29 B. M. E. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000. to appear.

5.30 R. Niedermeier, K. Reinhard, and P. Sanders. Towards optimal locality in mesh-indexings. In *Proceedings of the 11th International Conference on Fundamentals of Computation Theory (FCT'97)*. Springer Lecture Notes in Computer Science 1279, pages 364–375, 1997.
5.31 K. R. Popper. *Logik der Forschung*. Springer-Verlag, Heidelberg, 1934. English Translation: *The Logic of Scientific Discovery*, Hutchinson, 1959.
5.32 W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2. edition, 1992.
5.33 J. O. Rawlings. *Applied Regression Analysis: A Research Tool*. Wadsworth & Brooks/Cole, 1988.
5.34 P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.
5.35 P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 849–858, 2000.
5.36 C. Schaffer. *Domain-Independent Scientific Function Finding*. Ph.D. thesis, Department of Computer Science, Rutgers University, 1990.
5.37 Computer Science and Telecommunications Board. Academic careers for experimental computer scientists and engineers. *Communications of the ACM*, 37(4):87–90, 1994.
5.38 R. Sedgewick. Analysis of shellsort and related algorithms. In *Proceedings of the 4th European Symposium on Algorithms (ESA'96)*. Springer Lecture Notes in Computer Science 1136, pages 1–11, 1996.
5.39 D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–33, 1958.
5.40 J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, Heidelberg, 1993.
5.41 W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.
5.42 J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, MA, 1977.
5.43 B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Symposium on Foundations of Computer Science (FOCS'99)*, pages 131–140, 1999.
5.44 L. Weisner. *Introduction to the Theory of Equations*. The MacMillan Press Ltd., London, 1938.
5.45 M. A. Weiss. Empirical study of the expected running time of Shellsort. *The Computer Journal*, 34(1):88–91, 1991.

6. WWW.BDD-Portal.ORG: An Experimentation Platform for Binary Decision Diagram Algorithms

Christoph Meinel[1], Harald Sack[1], and Arno Wagner[2]

[1] University of Trier, Dept. of Computer Science, Trier, Germany
{meinel,sack}@ti.uni-trier.de
[2] Institute TIK, ETH Zürich
Gloriastr. 35, CH-8092 Zuerich, Switzerland
wagner@tik.ee.ethz.ch

Summary.

There is an upcoming need for World Wide Web portal sites to facilitate access to resources for specific research communities. The portal site described in this paper provides a testbed functionality besides additional information resources that are of interest in the research in Binary Decision Diagrams (BDDs). In the last decade, BDDs have proven to be the state-of-the-art data structure in computer aided design of integrated digital circuits. To assess the strengths and weaknesses of manipulation algorithms for BDDs, benchmark calculation is one of the most important methods in BDD-research. Due to the inherent high sensitivity of these algorithms to the particular experimental setup it is rather difficult or even impossible to reproduce benchmark results for comparison or for independent result verification. We have designed and implemented a WWW based BDD-testbed that overcomes these problems and greatly facilitates BDD algorithm comparison.

6.1 Introduction

Today the Internet offers the possibility of standardized and global communication without a need for special hardware or expensive infrastructure. While the World Wide Web (WWW) has become the largest information resource, esp. also for sciences and research, its inherent lack of any structure is responsible for the fact that the direct access to specific information is difficult. Search engines are still far from indexing even large fractions of the entire web [6.9] and even concerning the part being indexed the results of their searches are almost unstructured. Thus, in most cases, the user is faced to an all-or-nothing situation, where queries return even far too many hits or none at all.

6.1.1 WWW Portal Sites for Research Communities

One solution is the recent notion of specialized "portal"-sites that organize WWW-content into categories and in some cases grade the quality of the

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 127–138, 2002.
© Springer-Verlag Berlin Heidelberg 2002

provided information. But, most portal-sites are targeted to the general public and relatively small communities as, e.g. research communities are not commercially interesting to the existing portals. On the other hand, research oriented portal sites might also provide access to highly specialized research tools available to the research community via an appropriate web interface.

Today, in computer science, experimental evaluation of algorithms resulting from theoretical work has become more important. This comes due to the fact that more and more important algorithms, although having bad theoretical average-case properties due to their heuristic nature, perform well in practical applications.

6.1.2 Binary Decision Diagrams

Algorithms for Binary Decision Diagrams (in particular we address Ordered Binary Decision Diagrams, but they will be further simply referred to as BDDs or OBDDs) are a good example for that case. In computer aided design of very large scale integrated circuits BDDs have been established as the state-of-the-art data structure. They are extensively used for simulation, modeling, and verification of digital circuits, often being orders of magnitude more powerful than other techniques. For an overview of BDD related research, see [6.11].

But, while BDD-based methods perform well in many cases, the underlying problem of representing subsets of a Boolean vector space is known to be hard. This means that circuit descriptions given as a relatively small Boolean formula can have an extremely large BDD representation. Unfortunately, formulas and other alternative representation techniques are unsuitable for use in computations and at present it seems that BDDs are the most convenient representation for these purposes.

The important optimization algorithms for BDDs have exponential (time and memory) worst case complexity, but perform still better than any algorithm for alternative data structures in that particular area. To achieve meaningful results at reasonable expenses, heuristics are applied and for most applications they can be utilized to solve practical problems. But, the problem remains to get a meaningful assessment of the power of these heuristical methods. And for that reason, benchmark computations for comparing the qualities of different optimization heuristics are applied.

Algorithms for BDDs are rather sensitive to the details of the environment that is used for experimentation. Usual research papers will state something like "benchmark computation was done on a UltraSparc with 512MB of memory". While this will be adequate for some classes of algorithms and will allow reasonable predictions for the performance on other machines, it is insufficient for BDD benchmarking. Small details like a slightly different compiler or a different OS version can have a large impact on BDD performance. For that reason it is often impossible to verify published research results, sometimes to the extend that not even the authors of a paper are able to reconstruct

their results a year later without extensive reverse engeneering. To make the situation even worse, BDD algorithm performance is very much dependent on the exact nature of the computed benchmark. Performance figures for some benchmarks do not allow a reasonable prediction of the performance of a new heuristic for other types of benchmarks. This fact drastically limits the usability of published benchmark results and forces research groups to reimplement the heuristics they are interested in, in order to obtain their own benchmark results for their setting.

To address this problem, a finely standardized experimentation platform is required. Because of the necessary degree of accuracy, the only feasible way to achieve this, seems to be to provide a set of identical benchmark servers, where researchers can perform benchmark computations with their very own benchmarks. In this way they are able to evaluate new heuristics on their own circuits and verify other published results.

The authors have created a powerful and versatile experimentation environment that is fit for real world use and has actually been available since 1999. This environment offers access to a number of research tools that contain recent BDD algorithms. Questions of scheduling, load balancing, and error recovery have been solved in a satisfying manner within this system. It is embedded into a larger environment, a WWW portal site that supports the BDD research community in other ways as well.

When researchers do real world benchmark computations on systems other than their own, security and confidentiality questions arise. The authors have addressed these questions by allowing the use of encoded BDDs as input data format for benchmark computations. Because OBDDs have a canonical structure, circuit details are hidden. The purely Boolean function represented by an BDD is usually not a secret. Furthermore, this abstraction step does not reduce the meaningfulness of the computation as BDD-algorithms usually do this as a first step anyway.

An important step to gain acceptance with such an approach is the inclusion of methods developed by researchers other than the authors. The current system already features several heuristic methods provided by other researchers [6.6, 6.5, 6.2, 6.10], with the prospect that this number will grow.

6.2 A Benchmarking Platform for BDDs

When new BDD heuristics are developed, they are usually added to some existing software package. After optimization, when a sufficient level of performance is reached, the algorithm is presented to the community. Usually, evaluation is carried out by using benchmarks from a standard benchmark set first, for example the ISCAS 89 [6.4] set of circuits. Due to the nature of the problem, performance both in speed and memory consumption can vary extremely between different circuits of the benchmark. Predictions about the

behavior of a heuristic on other circuits are extremely difficult or even impossible. Because of this, there often arises the whish to allow other researchers to use the software, to evaluate the suitability of the new heuristic applied to their own problems.

6.2.1 To Publish Code is not Optimal

But, here the problem manifests itself. Let us consider giving away the source code of the implemented algorithm:

- The code usually only works within the software package it was written for.
- The code has research level quality and may still lack documentation or even worse, contain errors.
- Due to some ideas that are not yet published, one may not want to give away all the details of the implementation.

So, this is not an appealing option. But giving away executables means no improvement either:

- The software has to be adapted to different platforms.
- There is need for support for every other possible platform.
- There is need to develop the software to a higher level of stability.

Another option is reimplementation of the algorithm by interested parties. But this requires even larger effort and special experience. Almost nobody is willing to undertake such an effort for an uncertain outcome.

Further complications arise because the published results are difficult to reproduce as fine-tuning often is required and usually, it seems to be critical for good results. This makes comparisons outside of the published results extremely difficult. We think most of these issues can be addressed in a satisfying way by providing the possibility to access the tools containing heuristics by using the Internet.

We believe that some requirements need to be addressed to make an Internet based solution truly usable. Ease of use is achieved by using a WWW interface, but other aspects are important as well. The WWW Interface alone does not automatically improve the quality of the program code, but a well suited wrapper for external code might prevent a lot of possible ommisions and errors, which the authors did not pay attention in their coding. On the other hand, the problem of implementing the software within your own computing environment including all earlier mentioned error probabilities can be prevented. As the computations done are time and memory intensive (i.e. taking hours of cpu-time and up to hundreds of megabytes of memory), just writing some cgi-scripts clearly would represent an inadequate solution. A significantly larger effort is required, even if the system is only used for the evaluation of heuristics. Usability would be critically lowered if getting computations done would take very long or only very small "toy" examples could be computed.

6.2.2 What is Really Needed

Flexibility. It should be possible to add tools of various nature to the system with reasonable effort. We do not want to limit the system to a specific tool. If some researcher has a new heuristic and wants to publish it via our system, the amount of customization necessary should be as low as possible, and ideally, no more than a recompilation for the computing platform in use should be necessary.

Speed. To achieve an appropriate overall speed it is necessary to distribute the actual computations over several computers. The number of engaged computers should be easily adjustable, and it should even be possible to include computers that are remote and only reachable via an Internet connection. To maintain comparability of the results, it is mandatory that the pool of computers can be divided into groups of machines with comparable computing power.

Reliability. As computations can take hours and as there might be a number of still pending computations, there should be a mechanism that allows automated crash recovery without loss of submitted requests.

6.3 A Web-Based Testbed

With OHO (for OBDD Heuristics Online) we have developed a testbed environment that meets the requirements mentioned above [6.12].

6.3.1 The WWW Interface

The system is accessed by a web interface. After an introductory page containing general information, the user can access a menu that allows the selection of submission forms for individual tool and heuristic types, as well as browsing the specific documentation. When one of the possible types of computation is selected, the user is guided to a submission form, where all the relevant information needed for a first meaningful evaluation of an algorithm can be enterd. This includes an (optional) e-mail address for notification about results, tool options and selection of input data. Options include the choice between the featured heuristic for reordering or some reference heuristic. Input data can be provided by either, some predefined circuit, or by a circuit description file transferred by the user. File upload is done with the browser, a feature available with most ordinary Web browsers.

After the submission the request is queued and computations begin as soon as the required resources will become available. If a notification email address has been provided, the results will be delivered immediately after completion.

6.3.2 Implementation

The basic structure of the system is simple. Individual tools have individual user interfaces that have the task of accepting new requests, giving feedback about the status of a computation submitted previously, and returning the results to the submitter. A central scheduler manages every submitted request during its whole life-cycle. A group of computers perform the actual computations necessary to complete a request. If a computation is terminated because of unavailable resources due to local usage of the computer it was scheduled on, the request will be rescheduled on an other computer or restarted later. We use Linux as operating system.

To achieve **flexibility** every contained tool is fitted with a small wrapper script that controls its I/O. As long as the tool communicates via command line, processing the standard input and output with files, the customization of these scripts is quite simple. We believe that this approach covers the majority of research tools.

Speed is achieved by distributing the computations, as mentioned earlier. This distribution is managed by the central scheduler in such a way that non-dedicated computers can be used. Queuing of requests is performed here as well.

As requests will be queued and queues might get longer, **reliability** becomes an important issue. In order to achieve this, the scheduler performs frequent dumps of the current system status. In case of a crash the scheduler will automatically recover with the help of these dump files and will resume computations without the need for manual intervention.

6.3.3 Available BDD Tools

At the moment we have integrated recently published heuristics added to nanotrav (part of the CUDD system, [6.17]) by a number of researchers. These heuristics are not part of the CUDD standard distribution itself, but rather actual research code provided by the developers of the specific heuristics. There are as well some heuristics that are part of the CUDD standard distribution for the purpose of comparison. We also have heuristics online that were added recently to the well known model checker SMV [6.15]. At the moment, only new heuristics for model checking are provided that have been implemented by our research group. Additionally we had made available another online BDD heuristic from external researchers for the verification and synthesis tool VIS [6.3], before it was included into the last recent release of VIS.

Of course, we intend to incorporate many more tools and heuristics in the near future and therefore, we invite every researcher, who plans the publication of a specific new heuristic or any new tool within our portal to contact us. As stated above, we do not need access to source code and the addition of usual research tools will not be a problem at all.

6.4 Added Value: A BDD Portal Site

The WWW has the potential for becoming something like a "super library", allowing access to all information that somebody finds relevant enough to put up on some web pages. Today, its size and growth are impressive and the basic technology for accessing the WWW and maintaining a presence on the WWW seem to be adequately developed. But, by its very nature as a decentralized, somehow "anarchistic" medium, *structure* is the important point that is missing within the WWW. Despite the existence of sophisticated search engines it is often still difficult to find specific content. If someone only relies on traditional search engines the situation is likely to get even worse [6.9], and especially so for research content of limited public interest. We will discuss some of the reasons now.

6.4.1 Structure of a Conventional Portal

By a *conventional portal* we mean a general search site, as, e.g. *Yahoo!* [6.18] that supports general (raw) searches and in most cases has a set of prearranged categories where handpicked sites are accessible in a more structured way. Figure 6.1 depicts the basic structure of such a conventional portal site. The main information gathering mechanism is shown to the right: A web-robot or "spider" that searches known pages within the WWW for embedded links to other pages and includes the pages linked into the database of known pages. Usually, there is also some manual insertion of appropriate starting points for these searches, when new popular sites become online. As a third

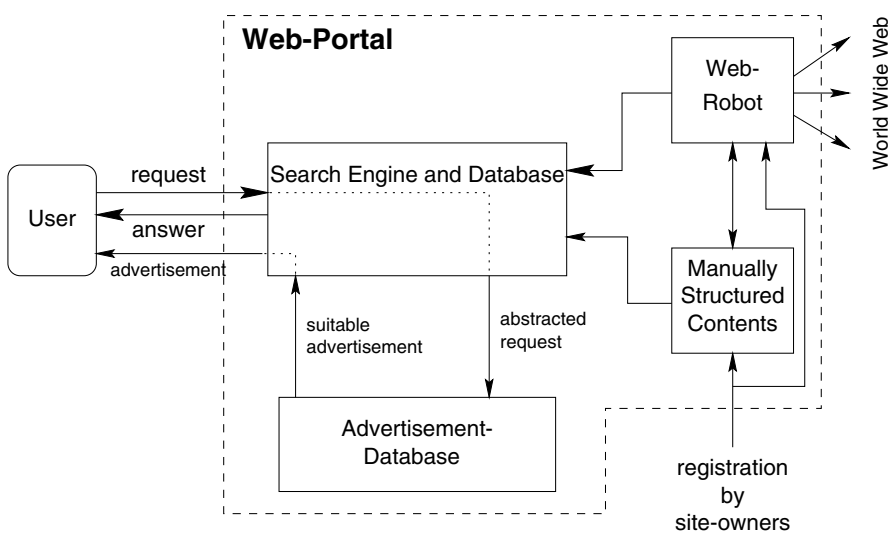


Fig. 6.1. A conventional portal

mechanism to insert new pages into the database, most conventional portals also offer the possibility that pages might be registered by someone, which usually means the site owners.

The central component of a conventional portal is the database including more or less sophisticated its search mechanisms. This component processes the search requests submitted by the users. In most cases, there is also a database of commercial advertisements that are sent back with the pages found for the users requests. Often additional effort is spent to match the advertising to the request the user has made.

6.4.2 Shortcomings of Conventional Portals

Here exactly, the inadequacy of a conventional portal manifests itself: The primary motivation for operating a conventional portal is to earn money. This is achieved by being paid for advertisements sent to the user together with the answers retrieved for his query. Hence, the main focus is to attract as many people as possible that are interested at least in some of the advertising delivered by the portal. And here the time for manual improvement of the content will be spent. Web pages of interest only to a small group, as e.g. researchers in a specific field, will usually only be added to the database if they are registered manually by their owners. With the number of conventional portals out there, registering pages most times requires a significant amount of work and additional maintenance.

For these reasons we believe that there is need for small, specialized portal sites that support specific research communities. These specialized portals should contain a significant amount of structured and preselected contents aimed at the specific needs of the addressed research community.

6.4.3 The BDD Portal

The structure of our BDD Portal can be found in Figure 6.2. The topmost component consists out of a collection of research oriented links. This includes a list of links to homepages of active researchers. "Active" in that particular sense means that they have to have at least one publication in the area of Binary Decision Diagrams. This list aims to be a complete representation of the BDD research community. For researchers, where no homepage could be found, the email address is provided instead. Additionally, links to specific BDD oriented research projects and working groups are gathered, and not forget to mention links to companies that are commercially promoting BDD technology and research.

The second component is serving as a datebook for all important events concerning BDDs, like conferences and workshops, including all relevant information as dates, deadlines and links to the according homepages. This collection is subject to permanent update.

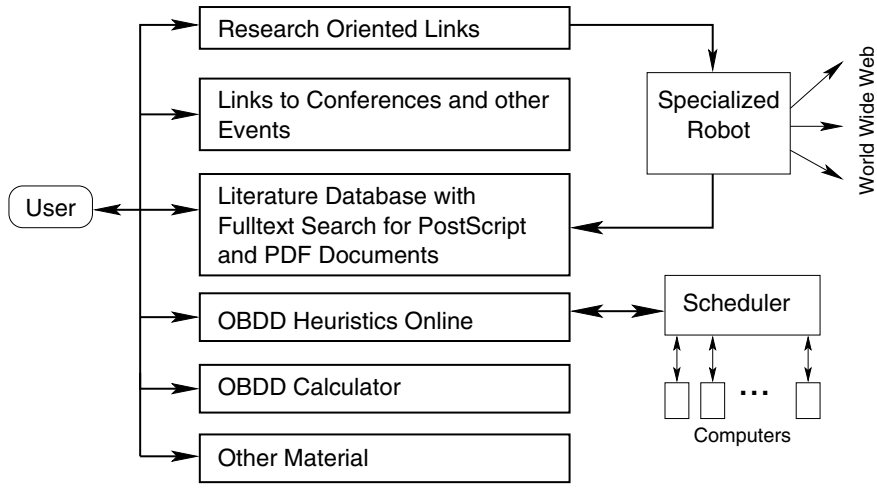


Fig. 6.2. The BDD Portal

The third component comprises a database of literature on BDDs available on the WWW. This database contains information about technical reports, papers and other texts, which can be searched in their full text, even if the documents are only available in Postscript or PDF (portable document format). Search results will include the first lines of text for documents found and links to their original location. The documents for this database are periodically collected by a specialized robot, that is driven by the links to homepages of researchers. The robot is capable of recognizing document formats and creates a searchable index out of them. This component represents a significant improvement compared to conventional search engines, which, in most cases are not able to access the contents of non-html documents. This database is endorsed by a collection of links to journals, where articles on BDDs are published, specific series of technical reports related to the subject, and, as well, by links to available monographs and textbooks. Additionally, links to lecture notes and course materials of several international universities are provided and maintained.

The fourth component, OBDD Heuristics Online, is our effort to address some of the specific problems with the publication of results from research in heuristics for BDDs, which was explicitely described in the previous section.

In addition to these four components, a tool providing the possibility to test and to visualize BDD computations is added, the OBDD Calculator. The OBDD Calculator offers the possibility to perform BDD manipulation operations and symbolic simulation of formula input or circuit descriptions. It serves as a graphical front-end to the CUDD BDD-package and provides the possibility of a graphical visualization of the computed BDDs. In addition to the BDD Calculator, a recently developed visualisation tool for BDD

algorithms is to be included, which offers the possibility to gain insight into the BDD synthesis algorithm. These features might be particularly useful for students to understand BDD algorithms and BDD manipulation operations, especially also for the purpose of tele-teaching. The user has the possibility to access a personalized database of circuit descriptions and BDDs, which can be subject to further research and studies during a course schedule.

Other material, as e.g. the links to relevant benchmarks, links providing download access to the relevant BDD packages, model checkers, and also related tools as SAT-solvers are added to the site as well.

6.5 Online Operation Experiences

OHO was released to the WWW in 1999 and soon started to raise the interest of the BDD research community. Besides the basic idea of maintaining an independent WWW-based platform for benchmark and algorithm evaluation for BDDs, the requirement for a centralized archive site dedicated to BDD research soon became obvious and the efforts of our working group were driven towards that desired goal.

In December 1999 the portal site `www.bdd-portal.org` went online and page requests starting at about 8000 pages a month, now, have reached 30000 and more page requests. This seems to be not much, compared to portal sites addressing the general public. But, for our small research community, the number is quite impressive. Several publications [6.12, 6.13, 6.14] about OHO and the related BDD portal site point out the importance of this contribution for the BDD research community on the one hand, and also for the development of specialized portal sites, providing the benefit of information structure to the WWW on the other hand.

More resources and links for BDDs will be added to our portal site, in order to further establish this portal as a central announcement and link site for all information in the WWW connected to BDDs. We are thinking about a mechanism that allows remote administration of the conference and workshop database by the organizers of the events.

More Decision Diagram heuristics and tools will be added to the section that allows online evaluation. In fact the experimentation platform should be kept current with ongoing research by our efforts as well as by external contributions. There might also be other kinds of interfaces than the WWW interface that would prove beneficial.

The portal can be accessed at `http://www.bdd-portal.org`.

6.6 Related Work

As far as the authors know, there is no other effort to create a specialized portal site for BDD research and there are only very few sites targeted at

specific research communities. The authors also do not know of any effort to make BDD-tool functionality available via the WWW on a comparable scale. There seem to be a few efforts capable of computing only small examples, e.g. the WWW interface for the word-level DD package developed by Stefan Höreth [6.7]. There, only small Boolean functions in terms of Boolean formulas can be transformed into several different types of decision diagrams that can be visualized in a way similar to the BDD Calculator of our BDD portal.

There have been efforts to use the WWW as an unified interface to a heterogenous set of EDA (Electronic Design Automation)tools. In [6.1] such an application has been described. The main focus there is the integration of different tools on different platforms with different data formats into a seemingly homogenous environment. The framework is intended to do actual work rather than to allow the comparison of the power of different approaches to a specific problem (like BDD heuristics) as in our case.

In [6.8] the vision of integrating a great number of different EDA CAD services running in different places into one "global" EDA CAD system using the Internet has been described. The main techniques here are the use of proxies to abstract the actual tool being used. In [6.16] another approach to the integration of different EDA tools is mentioned. This approach is mostly centered on the notion of "active messages".

References

6.1 A. Bogliolo, L. Benini, G. De Micheli, and B. Riccó. PPP: A gate-level power simulator — a World Wide Web application. Technical Report Stanford Technical Report No. CSL-TR-96-691, Stanford University, 1996.

6.2 B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Proceedings of IWLS'95*, Lake Tahoe, 1995.

6.3 R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincenteli, and F. Somenzi. VIS: a system for verification and synthesis. In *Proceedings of Computer-Aided Verification*. Springer Lecture Notes in Computer Science 1102, pages 428–432, 1996.

6.4 F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1929–1934, May 1989.

6.5 R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable reordering of OBDDs. In *Proceedings of IWLS'95*, Lake Tahoe, 1995.

6.6 H. Higuchi and F. Somenzi. Lazy group sifting for efficient symbolic state traversal of FSMs. In *Procceedings of ICCAD'99*, pages 45–49, 1999.

6.7 S. Höreth. A word-level graph manipulation package. *Software Tools For Technology Transfer*, 3(2):182–192, 2001.

6.8 D. Ku. Embedded tutorial: EDA and the network. In *Procceedings of ICCAD'97*, 1997.

6.9 S. Lawrence and C. L. Gilles. Accessibility of information on the web. *Nature*, 400(6740), 1999.

6.10 C. Meinel, F. Somenzi, and T. Theobald. Functional decomposition and synthesis using linear sifting. In *Proceedings of ASP-DAC'98*, pages 81–86, 1998.

6.11 C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.
6.12 C. Meinel and A. Wagner. OBDD heuristics online. In *Proceedings of IWLS'99*, Granlibakken, USA, 1999.
6.13 C. Meinel and A. Wagner. WWW.BDD-PORTAL.ORG. In *Proceedings of IWLS'00*, Dana Point, CA, pages 349–353, 2000.
6.14 C. Meinel and A. Wagner. WWW.BDD-PORTAL.ORG — an electronical basis for cooperative research in EDA. In *Proceedings of CRIS'00*, Helsinki, Finland, 2000.
6.15 CMU School of Computer Science. Formal Methods — Model Checking. `http://www.cs.cmu.edu/~modelcheck/`.
6.16 M. J. Silva and R. H. Katz. The case for design using the World Wide Web. In *Procceedings of the 32th Design Automation Conference*, 1995.
6.17 F. Somenzi. Colorado University Decision Diagram Package. `ftp://vlsi.colorado.edu/pub/`, 1996.
6.18 `http://www.yahoo.de`.

7. Algorithms and Heuristics in VLSI Design

Christoph Meinel and Christian Stangier

University of Trier, Department of Computer Science, Trier, Germany
{meinel,stangier}@ti.uni-trier.de

7.1 Introduction

The increasing complexity of nowadays VLSI designs makes it hard up to impossible to check their correctness by using validation methods like simulation. Therefore there is is a growing demand for formal verification methods in VLSI design and verification.

This paper presents application of heuristics in the field of symbolic formal verification.

In 1986 Bryant introduced ordered binary decision diagrams (OBDDs) that are still one of the most common data structures used for the verification of digital circuits. The use of OBDDs made it possible to circumvent two major problems in formal verification:

– exponential blow up of the circuit representation, and
– state space explosion in finite state machine traversal.

But it turned out that all optimization problems needed for OBDDs to work efficiently are at least NP-hard. For some problems it is shown that even approximation schemes do not exist. Thus, only heuristic approaches remain applicable. We will present heuristic approaches to some basic problems in OBDD based formal verification, i.e., variable reordering and partitioning of transition relations.

Since we apply our heuristics to intractable optimization problems the quality of the approach can only be judged by the use of benchmarks. We will discuss the benchmarking problems for OBDD applications.

A meaningful evaluation of heuristic algorithms requires a suited experimentation environment. We will discuss issues as use of open source software and the influence of parameter settings.

This article is structured as followed. The next section gives basic definitions of OBDDs, their algorithms and explains the most important applications, where OBDDs are used. In Section 7.3 we will present and improved heuristic for the variable reordering Problem. In Section 7.4 we will give algorithms for a typical OBDD-based application, namely the partitioning of transition relations.

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 139–162, 2002.
© Springer-Verlag Berlin Heidelberg 2002

7.2 Preliminaries

7.2.1 OBDDs – Ordered Binary Decision Diagrams

In 1986, by introducing *ordered binary decision diagrams (OBDDs)*, Randall E. Bryant got ahead a fundamental step in the search for suitable data structures in circuit design [7.4, 7.6]. Bryant's OBDDs combine two advantages: the new established data structure is not only quite space efficient but can also be handled efficiently from the algorithmic point of view.

Definition 7.2.1. *An* Ordered Binary Decision Diagram *(OBDD) P for a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is a directed acyclic graph consisting of inner nodes labeled by Boolean variables and sinks labeled by the Boolean constants 1 and 0. Each inner node has two outgoing edges: the 1-edge and the 0-edge. The OBDD has a starting node called root. The computation of $f(a_1, \ldots, a_n)$ follows a path from the root to a sink, where on a node labeled by x_i the input bit a_i is tested. If $a_i = 1$, the path follows the 1-edge, otherwise the 0-edge. The value of the reached sink determines the value of $f(a_1, \ldots, a_n)$. On a path from the root to the sink, each variable occurs at most once. The variables on a path respect a given order, which is (possibly after renaming) $x_1, \ldots, x_n$. For an edge leading from a node labeled by x_i to a node labeled by x_j it follows that $j > i$.*

An OBDD with more than one root node (i.e., representing $f : \{0,1\}^n \to \{0,1\}^m, m > 1$) is called a *shared* OBDD. In practice all functions to be represented are kept in one single shared OBDD. For simplicity we stay with the term OBDD.

Figure 7.1 gives two examples for OBDDs for the Boolean function $f = bc + a\overline{b}\overline{c}$ w.r.t. the variable order $a < b < c$.

From the Shannon decomposition on can derive the first important property of OBDDs:

Property 7.2.1 (Universality). Any Boolean function can be represented by an OBDD w.r.t. any predefined variable order.

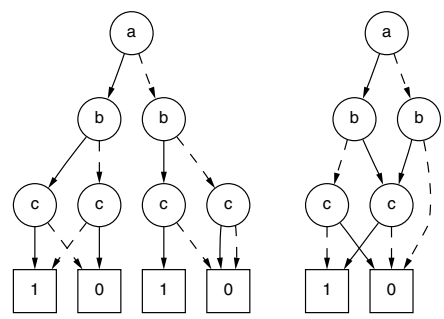


Fig. 7.1. Two OBDDs of $f = bc + a\overline{b}\overline{c}$

If we give up the restrictions on the variable order and the read-once property we get more general *decision diagrams*. Like in many other representations general decision diagrams have difficulties in handling Boolean functions in terms caused by the missing uniqueness, By using a surprisingly simple reduction mechanism, for OBDDs this problem can be solved very elegantly. Obviously, the following two reduction rules keep the represented function invariant:

Elimination rule: If 1- and 0-edge of a node v point to the same node u, then eliminate v, and redirect all incoming edges to u.

Merging rule: All terminal nodes with a given label are merged to one node, redirect all incoming edges to this node. If the non-terminal nodes u and v are labeled by the same variable, their 1-edges lead to the same node and their 0-edges lead to the same node, then eliminate one of the two nodes u, v, and redirect all incoming edges to the remaining node.

Definition 7.2.2. *An OBDD is called reduced if none of the two reduction rules can be applied.*

It is easy to see that the right OBDD in Figure 7.1 is reduced. Regarding the algorithmic properties of reduced OBDDs, the following property of canonicity is of basic importance:

Property 7.2.2 (Canonicity). With respect to a fixed variable order, the reduced OBDD of a Boolean function f is determined uniquely.

Besides universality and canonicity, OBDDs have a third fundamental property, which makes OBDDs such a successful data structure for representation of Boolean functions: the efficiency in algorithmic manipulation.

7.2.2 Operations on OBDDs

OBDDs are the only data structure for the representation of switching functions, whose representation size is not exponential in the number of variables for all functions (like truth tables) and that has deterministic polynomial algorithms for all important operations.

In the following the runtime and space requirements for these operations are given ($|P_f|$ denotes the number of nodes in the OBDD P for the function f, which depends on n variables and $a \in \{0,1\}^n$). All OBDDs have to respect the fixed variable order π.

Satisfiability test: $(\exists a f(a) = 1)$ Runtime: $O(|P_f|)$
Equivalence test: $(f \equiv g)$ Runtime: $O(min(|P_f|, |P_g|))$
Evaluation: $(f(a))$ Runtime: $O(n)$
Composition: $(f \otimes g)$ Runtime and space: $O(|P_f| \cdot |P_g|)$
Replacement by function: $(f_{x_i=g})$ Runtime and space: $O(|P_f|^2 \cdot |P_g|)$

Minimization: (Find the minimum OBDD representation of f w.r.t. π) Runtime and space: $O(|P_f|)$

It is worthwhile mentioning that most operations (except synthesis and replacement by function) have time and space requirements linear in the size of the OBDD. So, together with an efficient implementation OBDDs form a powerful data structure.

Efficient Synthesis of OBDDs. By $\otimes$ we denote an arbitrary Boolean operation, e.g., the conjunction or the disjunction. In order to compute the OBDD P_h of $h = f \otimes g$ from the OBDD representations P_f and P_g of two functions f and g, one uses Shannon's decomposition w.r.t. the leading variable x in the variable order π:

$$h = f \otimes g \;=\; x\,(f|_{x=1} \otimes g|_{x=1}) \;+\; \overline{x}\,(f|_{x=0} \otimes g|_{x=0}),$$

where $f|_{x=1}$ is the subfunction that results from f after replacing the variable x by the constant 1. By repeated application of this decomposition an OBDD representation P_h of the function h is computed.

In an OBDD P_f every node represents a subfunction f' of f. If the node that represents f' is marked with x_i, its successors represent $f'|_{x_i=1}$ resp. $f'|_{x_i=0}$. For the representation of any subfunction in P_h two pairs of nodes from P_f and P_g have to be computed. If one would simply follow the Shannon-decomposition, where the number of computations doubles on each level, 2^n pairs would be computed (for n variables). But, only $(|P_f| \cdot |P_g|)$ different pairs exist Thus, recomputation of pairs has to be avoided, as different subfunctions may be represented by the same node.

The already computed results from earlier stages are being recalled from a *computed-table*. In this way, the originally exponential number of decompositions is now bounded by the product of the two OBDD-sizes.

To increase the usage of the computed table all synthesis operations are mapped to a single operation, the so called if-then-else operator (ITE):

$$ITE(f, g, h) = f \cdot g + \overline{f} \cdot h.$$

E.g. $h = f \cdot g$ maps to $h = ITE(f, g, 0)$. Because of the huge number of ITE operations during synthesis the computed table is usually implemented as a cache to reduce memory consumption.

Another helpful construction is the usage of a *unique-table* which holds in a hash-table all already represented nodes. Before a node is created it is checked in the unique-table whether an functionally equivalent node already exists. This technique implements the merging rule. Together with an immediate check for the elimination rule the constructed OBDDs are always reduced and the reduction operation becomes obsolete.

Construction of OBDDs: Symbolic Simulation. The process of constructing an OBDD is called *symbolic simulation* of the circuit to be represented. Symbolic simulation is based on the synthesis operation:

Starting with the (trivial) OBDD representations of the input nodes one constructs, in topological order, OBDDs for each gate from the OBDDs of the corresponding predecessor gates.

Of course, it may happen that the OBDDs of the circuits are quite large. However, many circuits of the real world inherently contain much structure – hence, the reduction rules of the OBDDs cause the graphs describing the circuit to remain small.

7.2.3 Influence of the Variable Order on the OBDD Size

The size of an OBDD and hence the complexity of its manipulation heavily depends on the underlying variable order. An example is shown in Figure 7.2. With respect to the variable order $a_1, b_1, \ldots, a_n, b_n$ the function

$$a_1b_1 + a_2b_2 + \ldots + a_nb_n$$

has an OBDD representation of linear size. For the variable order $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ however, the size of the OBDD grows exponentially in n. It can be shown that any order that separates the a-variables from the b-variables leads to an exponentially large OBDD.

The same effect occurs in the case of adder functions: Depending on the variable order, the OBDD-size varies from linear to exponential in the number of input bits. Other important functions, e.g., the multiplication of two n-bit numbers imply OBDDs of exponential size w.r.t. every variable order [7.5, 7.33].

Due to the uniqueness of the OBDD representation of a Boolean function f w.r.t. a given variable order, the only way to optimize the size of the OBDD representation for f is to find a suited variable order.

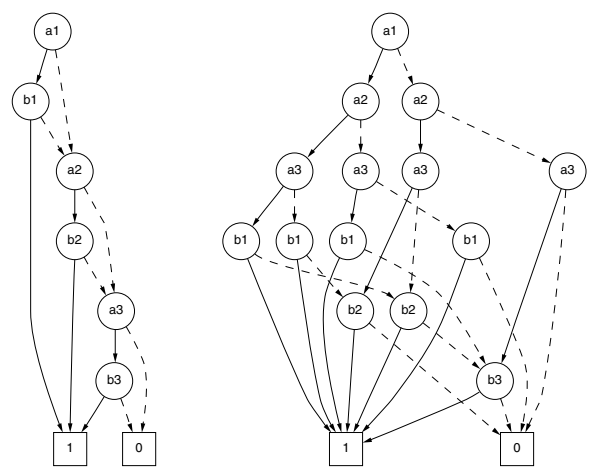


Fig. 7.2. Influence of the variable order

Optimization of the Variable Order. Due to the strong dependence of the OBDD-size upon the chosen variable order it is one of the most important problems in the use of OBDDs to construct "good" orders, i.e., orders that fit well to the represented function. However, the problem to construct an optimal order of a given OBDD is known to be NP-hard [7.30, 7.2]. The currently best known exact procedure is based on dynamic programming and has running time $O(n \cdot 3^n)$ [7.32]. Unfortunately, for real-life applications this method is useless. To make the problem even worse, Sieling [7.26] has shown that there in no polynomial time approximation scheme for the variable ordering problem unless P=NP.

Static Techniques. There exist a variety of heuristics to determine a variable ordering before building the OBDD of a function. These heuristics utilize various informations given by the netlist of the function [7.16, 7.22]. It turned out that these heuristics often work only for very specific functions. Nevertheless, these heuristics might be useful to determine starting orders.

Applications in OBDD based sequential verification often require representation of state sets, as these states sets and thus their OBDD representation changes a change in the variable order becomes necessary and a dynamic approach is required

Dynamic Techniques. *Dynamic Variable Reordering* is the process of improving the variable order and hence the size of an already built OBDD. Virtually, any optimization paradigm has been applied to variable reordering from genetic techniques to simulated annealing but one of the most successful strategies still is the local search algorithm proposed by Ruddell 1993. The so called *Sifting* [7.24] algorithm is based on the swap operation of two variables in the order, which can be carried out locally and hence is very efficient.

We will discuss Sifting in more detail in Section 7.3.

7.2.4 Reachability Analysis

Beyond verification of combinatorial circuits, sequential verification of finite state machines (FSMs) is the major field of application for OBDDs. An important task during verification of FSMs is the exploration of the system's state space.

BFS Traversal. Since the set of reachable states can be quite large, an explicit representation of this set, e.g., in form of a list, cannot be suitable under any circumstances. Coudert, Berthet and Madre have investigated the characteristic function of state sets which can be considered as a Boolean function and therefore be represented by an OBDD [7.9, 7.11]. They have shown that this representation form goes well together with the operations which have to be performed for the computation of the reachable states: If reachable states are computed according to a breadth-first-traversal then the representation via the characteristic function allows to compute all corresponding successor states within a single step. For this reason, one also uses

```
traverse(δ, q0) {
/* Input: Next-state function δ, initial set S0 */
/* Output: Set of reachable states */
    Reached = From = S0;
    do {
        To = Img(δ, From);
        New = To \ Reached;
        From = New;
        Reached = Reached ∪ New;
    } while (New ≠ ∅);
    return Reached;
}
```

Fig. 7.3. Basic algorithm for reachability analysis based on breadth-first traversal

the term *symbolic breadth-first traversal*. Once more, the complexity of the computation depends on the OBDD-size of the occurring state sets. For an outline of the traversal algorithm see Figure 7.3.

Image Computation. The computation of the reachable states is a core task for optimization and verification of sequential systems. The essential part of OBDD-based traversal techniques is the transition relation (TR):

$$\mathrm{TR}(x, y, e) = \prod_i \delta_i(x, e) \equiv y_i,$$

which is the conjunction of the transition relations of all latches (δ_i denotes the transition function of the ith latch, x, y, e represent present state, next state and input variables).

Partitioned Transition Relation. The transition relation is *monolithically* represented as a single OBDD and such a monolithic representation is usually much too large to allow an efficient computation of the reachable states. Therefore, more sophisticated reachable states computation methods make use of a *partitioned* TR [7.7], i.e., a cluster of OBDDs each of them representing the TR of a subgroup of latches. A transition relation partitioned over sets of latches $L_1, \ldots, L_j$ can be described as follows:

$$\mathrm{TR}(x, y, e) = \prod_j \mathrm{TR}_j(x, y, e) \text{ , where}$$

$$\mathrm{TR}_j(x, y, e) = \prod_{i \in P_j} \delta_i(x, e) \equiv y_i.$$

7.2.5 Image Computation Using AndExist

The reachable states computation consists of repeated image computations $Img(\mathrm{TR}, R)$ of a set of already reached states R:

$$Img(\mathrm{TR}, R) = \exists_{x,e}(\mathrm{TR}(x, y, e) \cdot R)$$

With the use of a partitioned transition relation the image computation can be iterated over P_i and the $\exists$ operation can be applied during the product computation *(early quantification)*:

$$Img(\mathrm{TR}, R) = \exists_{v^j}(\mathrm{TR}_j \cdot \ldots \cdot \exists_{v^2}(\mathrm{TR}_2 \cdot \exists_{v^1}(\mathrm{TR}_1 \cdot R)\ldots),$$

where v^i are those variables in $(x \cup e)$ that do not appear in the following $\mathrm{TR}_k, (i < k \leq j)$.

The so called *AndExist* [7.7] or *AndAbstract* operation performs the AND operation on two functions (here partitions) while simultaneously applying existential quantification $(\exists_{x_i} f = (f_{x_i=1} \vee f_{x_i=0}))$ on a given set of variables, i.e., the variables that are not in the support of the remaining partitions. Unlike the conventional AND operation the AndExist operation only has a exponential upper bound for the size of the resulting OBDD, but for many practical applications it prevents a blow-up of OBDD-size during the image computation.

Another important problem is finding an optimal schedule of the partitions for the AndExist operation. Geist and Beer [7.13] presented a heuristic for the ordering of partitions each representing a single state variable. The goal of this heuristic is to keep the support variable set of the intermediate products as small as possible. This heuristic was broadened by Ranjan et al. [7.23] to allow partitions including more than one state variable.

An insight into the complexity of the partition problem was given by Hojati et al. [7.14]: they have shown that finding a tree of conjunctions s.t. the support of the largest intermediate product is less than a given constant is NP-complete even under the simplifying assumption that the support of $f \wedge g$ is the union of the supports of f and g.

Symbolic Model Checking. Since a complete formal verification of a sequential system is often too complex, methods are of interest that guarantee at least correctness of certain properties. One of them is the so-called *model checking*.

Model checking is the problem to decide whether an implementation satisfies its specification given in terms of a temporal logic, e g., the so-called *computation tree logic* (CTL). The formulas of CTL describe properties of infinite pathes of states that are traversed during the computation.

The idea to use OBDDs for a symbolic representation of state sets during model checking was first introduced by McMillan [7.17] and Coudert/Madre [7.10]. Using this way of *symbolic* model checking, real-life systems up to 10^{100} states can be verified. For an introduction to model checking and CTL see [7.17].

7.3 Heuristics for Optimizing OBDD-Size — Variable Reordering

Most of the reordering heuristics published so far were general purpose algorithms independent on the application where they were used. This assured them universality, but on the other side, a lot of useful information was ignored. This section is devoted to an approach that uses the meaning of the functions represented by OBDDs in a particular application for speeding up the computational process. The main idea is to focus minimization on that part of the OBDD that represents the functions used in the next steps of the computation. We call these functions *key functions* and the corresponding subOBDDs *key OBDDs*. Obviously, the set of key functions is dynamically changing during the computation. If the size of the rest of the OBDD remains manageable, we achieve a gain for two reasons: the minimization of a part of a OBDD can be performed faster than for the entire OBDD, and secondly, the particular OBDD operations are faster. The latter is caused by the fact that the operations are performed over the key functions that have due to the approach smaller OBDD representation than they would have had if any usual reordering strategy aimed to the minimization of the whole OBDD would have been used.

Although the definition of the key OBDDs is dependent on an application, it requires only a small extension of the interface of the application software to the OBDD package used. The main part of the reordering can be implemented in the OBDD package itself. In this sense, the proposed method is universal and can be used in diverse applications. In the following we will describe an application of sampling to symbolic model checking.

7.3.1 Sample Reordering Method

Random sampling is a technique successfully used in several hard discrete problems. The idea to use sampling for the variable order problem arises naturally from the character of the problem.

The difficulty of the dynamic reordering problem does not arise from the size of the search space, but from the fact that the quality of the found solution (i.e., the OBDD size) can only be determined by constructing the resulting OBDD.

The first application of sampling to variable reordering was presented by Meinel and Slobodovà in [7.28]. The basic idea can be summarized in a few sentences: A part of the OBDD is chosen as a representative of the OBDD and the minimization problem is solved for this part. The new order found as a feasible (or even optimal) solution for the sample is extrapolated and applied to the entire OBDD. If the attempt is evaluated as successful, i.e., the reduction of the OBDD size achieved a given threshold value, the algorithm terminates. Otherwise further attempts with new samples are undertaken, until success, or the number of allowed attempts is exhausted.

Obviously, the choice of a sample has an influence on the variable order found. It can be done in random manner (like in many sampling strategies) or by use of some structural and semantic properties of the OBDD under consideration. In our approach, we use randomness, but we are targeting to key OBDDs, i.e., we chose random fractions of the key OBDDs. Randomness substitutes a missing information, assures a balance between the key OBDDs and the rest of the OBDD, and avoid repetition of the same sample choice.

There are several important implementation details that may play an important role on the success of the heuristic, e.g., how to minimize the sample, how to extrapolate the compute order, or how to rebuild the whole OBDD with respect to the new order. In the following we describe an implementation of the Sample Reordering in the CUDD package [7.29].

Let *InitialSize* be the OBDD size at the start of the reordering. A sample of the size

$$SamplePortion \times InitialSize,$$

is chosen from the OBDDs whose roots are passed by the application/user. If no roots are given random sampling is chosen. The chosen sample is copied to a new OBDD and then reordered by means of Sifting. The new variable order is derived from the new variable order of the sample. The variables that do not occur in the sample are kept on their old positions. All other variables are moved according to their positions in the new variable order of the sample. Rebuilding of the entire OBDD with respect to the new order is done by subsequent movement of each variable to its new position. We monitor the size of the OBDD during this reconstruction. If there is a better order with respect to the corresponding OBDD size than the target order, we shuffle variables back to this order. This is also the case of an unsuccessful attempt when the new order is worse than the original one. The process of the rebuilding is interrupted if the size of the OBDD grows over a given threshold:

$$ChangeOrderBound \times InitialSize,$$

This may happen even if the targeted order is better than the original but the peak size is too big.

The CUDD package has a user option for grouping of variables. Variables in a group should be always kept together. This is useful for some applications where the meaning of variables is known and can be used as a navigation in the search for a good order, e.g., the couple of present and next state variable in the coding of finite state machines. If such groups are defined, they are respected by the new order, too. The rebuilding procedure moves the variables of the same group together. Also the candidates for a better order that could be found during the rebuilding process are required to fulfill the group restrictions.

If the new size of the OBDD is less than

$$ExpectedReduction \times InitialSize,$$

the reordering is considered to be successful. Otherwise, a second attempt with a new sample is allowed.

7.3.2 Speeding up Symbolic Model Checking with Sample Sifting

Sample Sifting is a good candidate for speeding up model checking. But, a successful application of the sampling method to model checking is challenging: Any branch-and-bound algorithm has a trade-off between computation time and quality of the result. In model checking the problem arises from the fact that a poor order results in larger OBDDs that require more computation time. Also larger OBDDs lead to earlier and more time consuming calls to variable reordering. Thus, the trade-off multiplies and there are usually not enough calls to variable reordering to compensate these effects. Nevertheless, a successful sampling strategy for symbolic model checking can be implemented, if the following points are taken into account:

Sample Size. The size of the sample is the most important parameter of sample sifting. Choosing a smaller sample will reduce the computational overhead for copying the sample. But even more important: The accelerating effect of sample sifting results from the fact that only a small OBDD is reordered, also resulting in smaller intermediate OBDD sizes during the reordering. The smaller the sample is, the faster the reordering performs. But, the sample cannot be chosen arbitrarily small, because in this case it does not represent the original OBDD's properties sufficiently. The result of the reordering usually will be a poor ordering for the original OBDD. Thus, the size of the sample directly influences the quality of the computed order. To fulfill the quality requirements of model checking the sample has to be chosen larger than for combinatorial applications.

Method for Reordering the Sample. As stated above the time saved be sample sifting results from sifting a smaller OBDD. One may try to accelerate even this reordering, but this will usually result in variable orders of less quality. Instead, we suggest to reorder the sample even more by enlarging the search space, e.g., by allowing a larger growth of the OBDD during reordering. This may compensate the quality losses resulting from reordering only a fraction of the OBDD.

Number of Attempts per Reordering. More than one sampling attempt per reordering might be a good idea for combinatorial application but not for model checking for the following reasons:
– Due to the small number of reorderings, several trials will compensate all the time savings, especially if larger samples are used.
– In some situations OBDD sizes grow despite of good variable orders. Here any reordering will fail.

While the above points are about reordering time, the following points deal with the choice of the sample that is crucial for the quality of the computed order.

Sample Without Semantical Information. If no external semantical information is available one may at least use some structural information about the represented functions. One may use a random strategy *Random Sampling* proposed by [7.28]: Starting from the top level of the OBDD nodes that are not representing projection functions (i.e., $f = x_i$) are chosen randomly as roots of subOBDDs for the sample. This process is repeated level by level until the size requirements for the sample are fulfilled.

Another strategy is to chose the sample from the roots with the largest subOBDDs. Unfortunately, this strategy does not work well. Obviously, optimizing the order of only a few OBDDs does not meet the requirements of all represented functions.

Sample with Semantical Information. One should make use of the semantical information about represented functions provided by the model checker. In [7.28] it is proposed to use *recently-used-roots*, i.e., roots involved in operations in the last steps of the computation.

In more detail: The roots resulted from the Boolean operations are pushed into a stack. Any garbage collection of the unreferenced nodes is completed by cleaning the stack. The size of the stack is bounded. Its capacity can be set according to the considered application and examples. The push operation into a full stack discards the bottom item. When the sample reordering is invoked, the sample is preferably built from the roots in the stack. If the OBDDs whose roots are in the stack do not suffice to cover the requirements on the size of the sample, we choose additional roots randomly.

Again, this strategy is not suitable for model checking, since the huge number of operations will result in a random choice of roots. In [7.19] it was shown how to utilize the *key functions* of FSM traversal like the transition relation or the reachable state set to get a good sample for reordering. Here, we use *recently-used-important-roots*, i.e., roots involved in elementary model checking operations like Exist-Abstract, Universal-Abstract and And-Abstract (see [7.17]), since state sets play a minor role in model checking. If we cannot fulfill the size requirements for the sample by using important roots we fall back to Random Sampling. Using this strategy we obtain the best results for sampling.

Methods for Copying. In [7.28, 7.19] copying a fraction of an OBDD is done in the following way (postorder): The OBDD is traversed in DFS order and the nodes are copied to the sample whenever a node is backtracked. This is done until the required size of the sample is reached. This method copies at first the lower part of the OBDD. The resulting sample is a subfunction of the original OBDD. If only a small sample is chosen it will leave some variables of the upper part of the OBDD (see Figure 7.4a).

To avoid this, we use the following method (preorder): The OBDD is also traversed in DFS order. But, the nodes are copied to the sample when the node is visited the first time. This results in samples that include usually all variables and the outline of the sample is related to the outline of the original

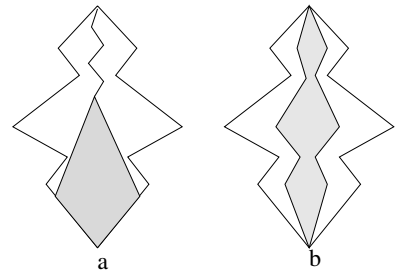


Fig. 7.4. a) Sample using postorder method b) Sample using preorder method

OBDD, i.e., from a level with many nodes a larger number of nodes is chosen for the sample. Applying this method results in unvisited edges that are set to the 1-sink. Thus, the resulting sample is modified but closely related to the original function (see Figure 7.4b). Our experience has shown that the preorder method works more stable and produces better results than the postorder method.

7.3.3 Experiments

We implemented our sifting strategies in the CUDD Package [7.29] (version 2.3.0). All experiments were performed on Intel PentiumPro 200MHz Linux Workstations with 250MByte datasize and CPU-time limited to 4 hours. For all computations we used the common technique of grouping present- and next-state variables, i.e., a present-/next-state pair is always kept in adjacent levels. This on the one hand accelerates reordering and on the other usually results in better orders. We compare our results to the standard sifting method.

For our experiments we used the publicly available SMV-traces of Yang [7.34, 7.35]. SMV is the description language for the SMV-model checker [7.17]. Traces are recorded calls of OBDD operations done by the SMV model checker during the computation of a certain model. With the use of Traces, one is not restricted to use the underlying OBDD-package of SMV instead one can use any OBDD package and/or own algorithms. The underlying SMV-models come from different sources and represent a range from communication protocols to industrial controllers. Traces have become the reference benchmark set for reordering during model checking. We used those traces, that require less than 250MB of memory and less than 4 hours CPU-time.

The choice of Traces as benchmarks enables us to show that our strategies are applicable to any OBDD based model checking tool and are not restricted to a special model checker.

Figure 7.1 gives an overview of computation time, reordering effort and peaknodes of the models computed with standard sifting as reordering strat-

egy. During reordering grouping of present- and next-state variables was enabled. The maximum allowed growth of the OBDD-size while sifting one variable was set to 20%.

The figure shows some evident differences of model checking in comparison to the OBDD application of combinatorial verification that mostly consists of symbolic simulation.. The number of variables (244 avg.) is comparable or even smaller than in combinatorial verification. The computation time is quite high (2044s avg.). The fraction of computation time, that is spent on reordering is extremely large (61% avg. of each reordering fraction), but only a few reorderings occur (4.7 avg.), while in combinatorial verification usually many reorderings occur. The average size reduction over all reorderings (avg. Size Reduction) is not very high. This results from the fact, that some reordering attempts do not result in smaller OBDDs at all (size reduction $<$ 5%). E. g. four reorderings during the computation of `furnace17` do no lead to smaller OBDDs, but one reordering drastically reduces the OBDD size (85%). Finally, the models are quite large (2.8 Mio. peaknodes avg.). Thus, most of them will not finish computation without reordering.

Results. Due to the random choice when copying a sample, for all experiments 10 single runs were performed.

For experiments we used the method *Important Roots* (IR) with sample size of 30% and 40%. For experimental results see Figure 7.2 and Figure 7.3. All samples are chosen by using the *preorder* method. We were able to decrease the average computation time up to 35% and the overall computation time up to 34%. The maximum improvement is 70%.

Since we obtained our results with a very loose coupling of the model checker to the OBDD-package, a tighter coupling to the model checker e.g., having exact knowledge about the represented functions would lead to even better results.

7.4 Heuristics for Optimizing OBDD Applications – Partitioned Transition Relations

The quality of the partitioning is crucial for the efficiency of the reachable states computation. The image computation is iterated over the partitions and includes costly computations. Therefore, maintaining a large number of partitions is time consuming. A small number of partitions may lead to unmanageable large OBDDs. One extremum of this trade-off is the partitioning where each latch forms a partition, which is usually small but requires many iterations. The other extremum is a monolithic transition relation (TR), that can be computed in one iteration but has large OBDD-size. Furthermore, the ordering of latches and clusters is crucial for an efficient AndExist operation. Using a poor order may lead to extremely large intermediate OBDD sizes that could make a complete image computation impossible.

Table 7.1. Overview of computation and reordering effort for the benchmarks using standard sifting

	Vars	CPU-time/s	Reorder-time/s	Re-ord.	avg. Size Red.	Reord. >5%	Peakn. in 1000
dartes	198	504	441 87%	3	8%	1	583
dme2-16	586	3757	2331 62%	5	18%	3	5151
dpd75	600	4574	2676 58%	5	0%	0	3296
ftp3	100	1119	588 52%	4	1%	0	3126
furnace17	184	3938	1328 33%	5	21%	1	2373
key10	140	846	643 76%	6	24%	3	1099
mmgt20	264	1610	860 53%	4	2%	0	2904
motors-stuck	172	265	142 53%	4	36%	3	670
over12	174	3002	2526 84%	6	7%	2	4725
phone-async	86	2604	1094 42%	5	8%	1	6118
valves-gates	172	268	200 74%	5	35%	5	542
sum	2686	22487	12829 57%	52	160%	19	30587
avg	244	**2044**	1166 **61%**	**4.7**	15%	1.7	**2781**

Table 7.2. Comparison of CPU-time for standard sifting and sample sifting

	Sifting	Sample Sifting			
Sample Size		30%		40%	
	time/s	%	time/s	%	time/s
dartes	504	+70	149	+62	194
dme2-16	3757	+45	2073	+53	1765
dpd75	4574	+28	3304	+ 9	4144
ftp3	1119	+43	635	+34	742
furnace17	3938	+41	2341	+35	2545
key10	846	+33	568	+28	610
mmgt20	1610	- 9	1770	-17	1961
motors-stuck	265	+44	147	+38	164
over12	3002	+51	1475	+39	1831
phone-async	2604	+13	2268	+13	2273
valves-gates	268	+24	202	+14	220
sum	22487	+34	14934	+27	16458
avg		+35		+28	

In the following we will describe the standard partitioning strategy, followed by a description of the *RTL partitioning heuristic*.

7.4.1 Common Partitioning Strategy

A common strategy for partitioning of the TR as it is used e.g., by VIS [7.3, 7.23] proceeds in three steps:

1. **Order latches**. First, the latches are ordered by using a benefit heuristic [7.13] that performs a structural analysis of the latches' transition

Table 7.3. Comparison of peaknodes in thousands for standard sifting and sample sifting

	Sifting	Sample Sifting			
Sample Size		30%		40%	
	nodes	%	nodes	%	nodes
dartes	583	-17	707	-17	707
dme2-16	5151	-12	5824	-13	5945
dpd75	3296	- 9	3633	- 8	3566
ftp3	3126	+ 4	2986	+10	2806
furnace17	2373	-16	2841	- 3	2439
key10	1099	-51	2236	-51	2236
mmgt20	2904	- 1	2945	- 1	2944
motors-stuck	670	-38	1073	-37	1058
over12	4725	+ 4	4550	+ 4	4543
phone-async	6118	- 7	6603	-24	8080
valves-gates	542	-43	950	-42	941
sum	30593	-11	34353	+13	35270
avg		-17		+16	

function to address an effective AndExist operation. During the iterated image computation next state variables are added while present state variables are quantified out. the benefit heuristic uses a greedy scheme to minimize the balance of introduced next state variables and quantified present state variables Additionally, the heuristic takes into account the highest index of a variable to be quantified out, resulting in a more efficient AndExist.

2. **Cluster latches**. The single latch relations are clustered by following a greedy strategy. Latches are added to an OBDD (i.e., by performing AND) until the size of the OBDD exceeds a given threshold.
3. **Order clusters**. In the last step the clusters are ordered similarly to the latches by using a benefit heuristic (VIS uses the same heuristic as in Step 1).

Figure 7.5a gives a schematic overview of this process.

7.4.2 RTL Based Partitioning Heuristic

Since modern complex designs require a structured hierarchical description to be feasible they are currently written in a hardware description language (HDL) at register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of *data flow* and *behavioral constructs*. Logic synthesis tools take the RTL HDL description to produce an optimized gate level netlist and high level synthesis tools at the behavioral level output RTL HDL descriptions. Verilog [7.31] and VHDL [7.15] are the most popular HDLs used for describing the functionality at RTL. Within the

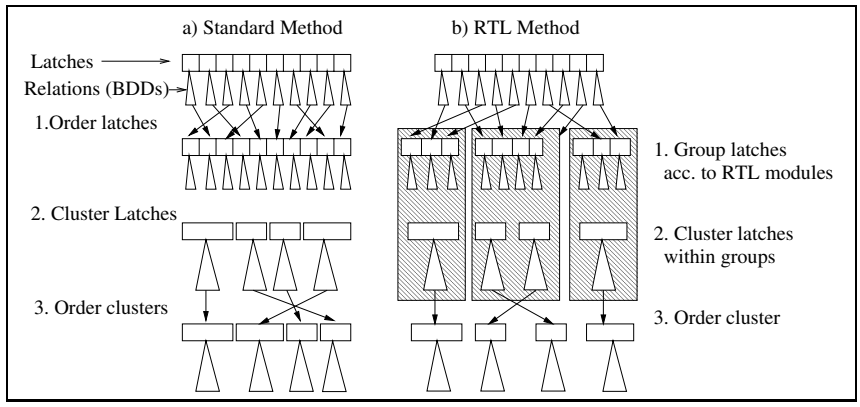


Fig. 7.5. Schematic of partitioning strategies

design cycle of optimization and verification the RTL level is an important and frequently used part.

The design methodology in Verilog is a top down hierarchical modeling concept based on modules, which are the basic building block. The experimental work for the following heuristic based on designs written in this language, but our approaches can be easily extended to any HDL or hierarchical FSM representation as it is, e.g., provided by state space decomposition algorithms (see, e.g. [7.18]).

As mentioned above, the way to build a complex design is to break it into modules, each with a dedicated functionality and a smaller complexity. For example communication protocols contain transmitters and receivers that represent independent modules. These modules are usually not too complex, thus the complexity of their TRs will be small. If a partition contains state variables of several modules, we need to represent the Cartesian product of these modules leading to a much more complex TR. The main reason for the efficiency of the partitioned TR approach is that state variables not appearing in other partitions are quantified out during the AndExist operation. This leads to much smaller OBDD-sizes and a faster computation. If the state variables of a module are spread over several partitions, the quantification does take effect only late in image computation. Therefore, most of the computation has to be done with large OBDDs.

RTL level description languages like Verilog [7.31] or VHDL [7.15] support a hierarchical design methodology by providing module constructs. As it can be seen this modularization has effects on the image computation that should not be neglected.

Although the standard method optimizes the partitioning twice, its main disadvantage is that it only uses structural information to optimize the partitioning for an efficient order for the AndExist operation during the image computation.

The RTL heuristic improves this optimization by including additional semantical information about the represented functions. As the experimental results show, there is a close connection between the RTL description and an efficient image computation.

The RTL heuristic proceeds in three steps:

1. **Group latches**. The latches are grouped according to the modules given in the top module of the RTL description in Verilog. Within the groups the latches are ordered by a lexicographic order that takes into account submodule names and bit numbers (names of latches from submodules are prefixed by the submodule name). Also, the bits of a certain register are named by the register and the bit number. The effect of this sorting is, that latches of a submodule within the group stay adjacent, without being grouped explicitly. The same holds for the bits of a register.
2. **Cluster groups**. The groups represent borders for the clusters. There is no cluster containing latches from different groups. To control the OBDD size of the clusters, the greedy partitioning strategy is applied within the groups. The clustering given by the groups lowers the influence of the arbitrary clustering produced by the OBDD-size threshold. Thus, resulting in a more *natural* partitioning.
3. **Order clusters.** (optional) In the last step the clusters may be ordered by using the benefit heuristic from the standard method.

Figure 7.5b gives an overview of this strategy.

Modifications of this strategy are possible:

- **Step 1a)** As an additional step the benefit heuristic of the standard method may be applied to order the latches within the single groups. It emerged that in our case the lexicographic order of the latches preserves more of the structure of the design and leads to better results.
- **Step 2a)** One may allow to create clusters that cross a group border. This will lead to a more compact representation of the TR with fewer clusters. Although the representation is more efficient the image computation does not perform as efficient as with the strict group borders.

7.4.3 Experiments

We implemented our strategy in the VIS-package [7.3] (version 1.3) using the underlying CUDD-package [7.29] (version 2.3.0). VIS is a popular verification and synthesis package in academic research. It inherits state of the art techniques for OBDD manipulation, image and reachable states computation as well as formal verification techniques. Together with the vl2mv translator VIS provides a Verilog front-end needed for our heuristic.

For our experiments we used Verilog designs from the Texas97 benchmark suite [7.1]. This publicly available benchmark suite contains real life designs including:

– MSI Cache Coherence Protocol
– PCI Local BUS
– PI BUS Protocol
– MESI Cache Coherence Protocol
– MPEG System Decoder
– DLX
– PowerPC 60x Bus Interface

The benchmark suite also contains properties given in CTL formulas for verification.

We left all parameters of VIS and CUDD unchanged. The most important default values are:

– Partition cluster size = 5000
– Partition method for MDDs = inout
– OBDD variable reordering method = sifting
– First reordering threshold = 4004 nodes

The reachable states computation or the model checking was preceeded by an explicitely triggered variable reordering. The CPU time was limited to 2 CPU hours and memory usage was limited to 200MB. All experiments were performed on Linux PentiumIII 500Mhz workstations.

Results. For results see Table 7.4 and Table 7.5. Img.comp. is the sum of all image and pre-image computations performed during the analysis. Part gives the number of partitions of the transition relation. The OBDD-size of the transition relation cluster and the peak number of live nodes is given by TRn resp. Peakn. The CPU time is measured in seconds and given as Time. The columns denoted with % describe the improvement in percent[1].

At the bottom of Table 7.5 you can find the sum of all numbers of partitions, BDD-sizes and CPU-times. Also, the *average of the relative improvement* is given as well as the *total improvement*

The experiments show significant improvements in time and space: The overall CPU time decreased by 67% overall and 40% on average. The method outperforms the standard method in 45 of the 47 benchmarks. The decrease in computation time ranges up to 90%. The OBDD peak sizes could be lowered by 62% overall and 25% on average. Interestingly, the RTL method results in 5% less partitions without requiring more OBDD nodes for the transition relation. This also proves the improved quality of the partitioning.

7.5 Conclusion

In this article we have presented algorithms for OBDD-based formal verification. All important problems for optimizing the OBDD data structure or

[1] $0 <$ improvement < 100; $-100 <$ impairment < 0.

Table 7.4. Comparison of original VIS partitioning, and RTL heuristic

	Img.-	Standard-VIS				RTL Method					
	comp	Peakn	Parts	TRn	Time	Peakn	%	Parts	TRn	%	Time	%
PClabnorm.PCI	304	176276	14	28613	253	145780	17	10	24054	16	157	38
PClnorm.PCI	206	81123	15	35124	56	69291	15	9	24472	30	52	7
TWO.contention	37	97623	7	11865	47	168938	-41	10	13207	-9	114	-58
multi_main.multim	45	38694	5	14700	33	33423	14	4	1588	89	18	45
p62_LS_LS_V01.ccp	64	166074	23	49952	200	141829	15	22	60455	-16	148	26
p62_LS_LS_V01.p6live	99	452267	23	49952	827	343308	24	22	60455	-16	367	56
p62_LS_LS_V02.ccp	53	146494	22	59487	107	121138	17	21	55439	7	65	39
p62_LS_LS_V02.p6live	96	167454	22	59487	181	139905	16	21	55439	7	120	34
p62_LS_L_V01.ccp	64	176540	23	49684	216	154852	12	21	61118	-18	141	35
p62_LS_L_V01.p6live	99	1614200	23	49684	3833	303656	81	21	61118	-18	384	90
p62_LS_L_V02.ccp	54	148560	23	62140	106	132040	11	22	55605	10	67	37
p62_LS_L_V02.p6live	89	183811	23	62140	193	136522	26	22	55605	10	118	39
p62_LS_S_V01.ccp	64	176540	23	49684	211	154852	12	21	61118	-18	136	35
p62_LS_S_V01.p6live	99	1614200	23	49684	3564	303656	81	21	61118	-18	401	89
p62_LS_S_V02.ccp	54	148560	23	62140	109	132040	11	22	55605	10	65	40
p62_LS_S_V02.p6live	89	183811	23	62140	199	136522	26	22	55605	10	114	43
p62_L_L_V01.ccp	52	164244	23	48961	194	142996	13	22	58344	-15	123	37
p62_L_L_V01.p6live	87	477543	23	48961	935	305267	36	22	58344	-15	244	74
p62_L_L_V02.ccp	53	144504	23	48971	180	131449	9	21	55967	-11	73	60
p62_L_L_V02.p6live	96	242452	23	48971	402	234951	3	21	55967	-11	214	47
p62_L_S_V01.ccp	75	168782	22	62479	121	134604	20	22	57345	8	120	1
p62_L_S_V01.p6live	118	192410	22	62479	231	241246	-19	22	57345	8	224	3
p62_L_S_V02.ccp	55	140767	22	57365	108	118826	16	22	51451	10	99	9
p62_L_S_V02.p6live	96	140767	22	57365	112	120239	15	22	51451	10	101	9
p62_ND_LS_V01.ccp	83	396642	24	63506	831	342628	14	23	61983	2	603	27

Table 7.5. Comparison of original VIS partitioning, and RTL heuristic cont

	Img.-	Standard-VIS				RTL Method						
	comp	Peakn	Parts	TRn	Time	Peakn	%	Parts	TRn	%	Time	%
p62_ND_LS_V02.ccp	103	191386	22	63321	356	166655	13	22	60733	4	230	35
p62_ND_LS_V02.p6live	192	1564426	22	63321	3922	847880	46	22	60733	4	1339	66
p62_ND_L_V01.ccp	75	356794	25	65964	824	323997	9	23	58708	11	616	25
p62_ND_L_V02.ccp	133	5860454	23	60383	>2h	1422284	76	23	57198	5	3352	54
p62_ND_L_V02.p6live	168	5573568	23	60383	>2h	990882	82	23	57198	5	1605	78
p62_ND_S_V02.ccp	84	150630	23	46744	187	157900	-4	23	61745	-23	135	28
p62_ND_S_V02.p6live	177	645918	23	46744	1221	363854	44	23	61745	-23	402	67
p62_S_S_V01.ccp	43	147063	23	62209	102	127339	13	21	59643	4	63	38
p62_S_S_V01.p6live	80	153012	23	62209	107	139405	9	21	59643	4	116	-6
p62_S_S_V02.ccp	37	129492	23	54800	96	116500	10	21	57223	-3	58	39
p62_S_S_V02.p6live	74	129492	23	54800	96	116500	10	21	57223	-3	59	38
p62_V_LS_V01.ccp	108	283494	24	58415	575	229601	19	23	58674	0	353	39
p62_V_LS_V01.p6live	114	4483034	24	58415	>2h	697992	84	23	58674	0	1290	82
p62_V_LS_V02.ccp	90	165200	23	52073	234	152143	8	21	58544	-10	143	39
p62_V_LS_V02.p6live	178	1059895	23	52073	2093	776538	27	21	58544	-10	832	60
p62_V_S_V01.ccp	82	213245	23	61795	258	200514	6	21	48349	22	244	5
p62_V_S_V01.p6live	127	964988	23	61795	2421	363716	62	21	48349	22	628	74
p62_V_S_V02.ccp	84	163439	22	54807	200	132192	19	22	56789	-2	134	33
p62_V_S_V02.p6live	177	351553	22	54807	515	250465	29	22	56789	-2	271	47
single_main.singlem1	108	14936	2	6352	13	9360	37	3	1470	77	8	41
2_proc.prop2	264	903917	4	12311	756	176315	80	3	2816	77	133	82
2_proc_bin.prop2_bin	140	252974	4	11610	154	84942	66	3	3663	68	43	72
Sum		31399248	952	2380850	49278	12236932	1186	903	2374640	270	16319	1896
Avg. of the rel. impr.							25%			6%		40%
Total improvement						**62%**		**5%**	**0%**		**67%**	

OBDD-based applications like reachability analysis are at least NP-hard. We have given two examples for heuristic approaches to the variable reordering problem and the problem of partitioning a transition relation. Both heuristics have been implemented in open source software packages and have been evaluated using public domain benchmark suites. This scheme is the only way to allow a fair judgment of new and improved algorithms in VLSI design.

References

7.1 A. Aziz et al. Texas-97 benchmarks. `http://www-cad.EECS.Berkeley.EDU/Respep/Research/Vis/texas-97`.

7.2 B. Bollig and I. Wegener, Improving the variable ordering of OBDDs is NP–complete. *IEEE Transaction on Computers*, 45(9):992–1002, 1996.

7.3 R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy and T. Villa. VIS: A System for Verification and Synthesis. In *Proceedings of the Symposium on Computer Aided Verification (CAV'96)*, pages 428–432, 1996.

7.4 R. E. Bryant, Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, pages 677–691, 1986.

7.5 R.E. Bryant, On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transaction on Computers*, 40:205–213, 1991.

7.6 R. E. Bryant, Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

7.7 J. R. Burch, E. M. Clarke and D. E. Long, Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on VLSI*, pages 49–58, 1991.

7.8 J. R. Burch, E. M. Clarke, D. L. Dill, L. J. Hwang and K. L. McMillan, Symbolic model checking: 10^{20} states and beyond. In *Proceedings of Logic in Computer Science (LICS'90)*, pages 428–439, 1990.

7.9 O. Coudert, C. Berthet and J. C. Madre, Verification of synchronous machines using symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Machines*. Springer Lecture Notes in Computer Science 407, pages 365–373, 1989.

7.10 O. Coudert and J. C. Madre, A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 126–129, 1990.

7.11 O. Coudert and J. C. Madre, The implicit set paradigm: a new approach to finite state system verification. *Formal Methods in System Design*, 6(2):133–145, 1995.

7.12 S. J. Friedman and K. J. Supowit, Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, 1990.

7.13 D. Geist and I. Beer, Efficient model checking by automated ordering of transition relation partitions. In *Proceedings of Computer Aided Verification (CAV'94)*, pages 299–310, 1994.

7.14 R. Hojati, S. C. Krishnan, R. K. Brayton, Early qantification and partitioned transition relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 12–19, 1996.

7.15 R. D. M. Hunter and T. T. Johnson, *Introduction to VHDL*. Chapman & Hall, 1996.
7.16 S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the 25th Design Automation Conference*, pages 6–9, 1988.
7.17 K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
7.18 I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan, and C. Pixley. Approximate reachability don't cares for CTL model Checking. In *Proceedings of the Internatuional Conference on CAD (ICCAD'98)*, 1998.
7.19 C. Meinel, K. Schwettmann, and A. Slobodová. Application driven variable reordering and an example implementation in reachability analysis. In *Proceedings of ASP-DAC'99*, Hongkong, pages 327–330, 1999.
7.20 C. Meinel and C. Stangier. Speeding up image computation by using RTL information. In *Proceedings of Formal Methods in CAD (FMCAD'00)*. Springer Lecture Notes in Computer Science 1954, pages 443-454, 2000.
7.21 C. Meinel and C. Stangier. Speeding up symbolic model checking by accelerating dynamic variable reordering. In *Proceedings of the IEEE 10th Great Lakes Symopsium on VLSI*, pages 39–42, 2000.
7.22 S. Minato, N. Ishiura, and S. Yahima. Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 52–57, 1990.
7.23 R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley, and B. Plessier. Efficient BDD algorithms for synthesizing and verifying finite state machines. Presented at the *International Workshop on Logic Synthesis (IWLS'95)*, 1995.
7.24 R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 42–47, 1993.
7.25 P. Savický and I. Wegener. Efficient algorithms for the transformation between different types of binary decision diagrams. *Acta Informatica*, 34:345–356, 1997.
7.26 D. Sieling. On the existence of polynomial time approximation schemes for OBDD minimization. In *Proceedings of the 15th International Symposium on Theoretical Aspects of Computer Science (STACS'98)*. Springer Lecture Notes in Computer Science 1373, pages 205–215, 1998.
7.27 D. Sieling and I. Wegener. Reduction of BDDs in linear time. *Information Processing Letters*, 48(3):139–144, 1993.
7.28 A. Slobodová and C. Meinel. Sample method for minimization of OBDDs. Presented at the *International Workshop on Logic Synthesis*, Tahoe City, CA, June 1998.
7.29 F. Somenzi. CUDD: CU Decision Diagram Package. `ftp://vlsi.colorado.edu/pub/`.
7.30 S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problem of shared binary decision diagrams. In *Proceedings of ISAAC'93*. Springer Lecture Notes in Computer Science 762, pages 389–398, 1993.
7.31 D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer, 1991.
7.32 I. Wegener. Branching programs and binary decision diagrams - theory and applications. *SIAM Monographs on Discrete Mathematics and Applications*, 2000.

7.33 P. Woelfel. New bounds on the OBDD-size of integer multiplication via universal hashing. In *Proceedings of STACS'01*. Springer Lecture Notes in Computer Science 2010, 2001.
7.34 B. Yang. SMV models. `http://www.cs.cmu.edu/~bwolen/software/`, 1998.
7.35 B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Proceedings of Formal Methods in CAD (FMCAD'98)*, pages 255–289, 1998.

8. Reconstructing Optimal Phylogenetic Trees: A Challenge in Experimental Algorithmics

Bernard M. E. Moret[1] and Tandy Warnow[2]

[1] Department of Computer Science, University of New Mexico
Albuquerque, NM 87131, USA
moret@cs.unm.edu
[2] Department of Computer Sciences, University of Texas
Austin, TX 78712, USA
tandy@cs.utexas.edu

Summary.

The benefits of experimental algorithmics and algorithm engineering need to be extended to applications in the computational sciences. In this paper, we present on one such application: the reconstruction of evolutionary histories (phylogenies) from molecular data such as DNA sequences. Our presentation is not a survey of past and current work in the area, but rather a discussion of what we see as some of the important challenges in experimental algorithmics that arise from computational phylogenetics. As motivational examples or examples of possible approaches, we briefly discuss two specific uses of algorithm engineering and of experimental algorithmics from our recent research. The first such use focused on speed: we reimplemented Sankoff and Blanchette's breakpoint analysis and obtained a 200,000-fold speedup for serial code and 10^8-fold speedup on a 512-processor supercluster. We report here on the techniques used in obtaining such a speedup. The second use focused on experimentation: we conducted an extensive study of quartet-based reconstruction algorithms within a parameter-rich simulation space, using several hundred CPU-years of computation. We report here on the challenges involved in designing, conducting, and assessing such a study.

8.1 Introduction

A *phylogenetic tree* or *phylogeny* is a representation of the evolutionary history of a collection of organisms, in which modern organisms are placed at the leaves of the tree and the (unknown) ancestral organisms occupy internal nodes; the edges of the tree thus denote evolutionary relationships. Due to difficulties in rooting the trees, these phylogenies are usually (but not always) represented by unrooted leaf-labelled trees. Figure 8.1 shows two proposed phylogenies, one for several species of the Campanulaceae (bluebell flower) family (from [8.10]) and the other for herpesviruses that are known to affect humans (from [8.8]). Note that the *Campanulaceae* tree is rooted through the use of a distantly related species (here tobacco), called an *outgroup* in this context (the root is taken to be the internal node to which the outgroup is attached); the herpesvirus tree is unrooted.

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 163–180, 2002.
© Springer-Verlag Berlin Heidelberg 2002

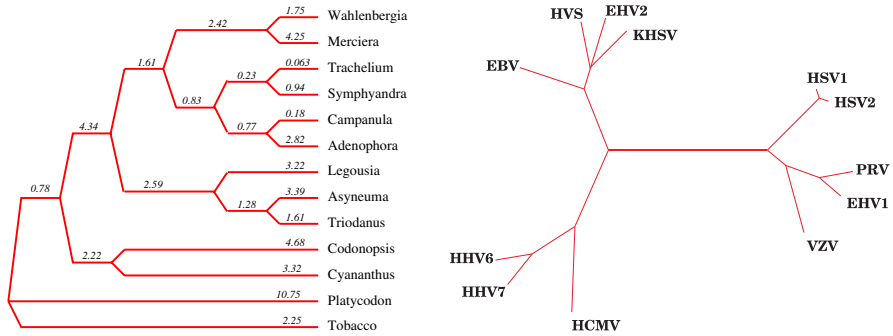


Fig. 8.1. Two phylogenies: some plants of the *Campanulaceae* family (left) and some herpesviruses affecting humans (right)

Reconstructing phylogenies is a major component of modern research programs in many areas of biology and medicine. An understanding of evolutionary mechanisms and relationships is at the heart of modern pharmaceutical research for drug discovery, is helping researchers understand (and defend against) rapidly mutating viruses such as HIV, is the basis for the design of genetically enhanced organisms, etc. In developing such an understanding, the reconstruction of phylogenies is a crucial tool, as it allows one to test new models of evolution. Due to the importance of phylogenetic trees in biological inquiry, there are many methods available for reconstructing phylogenetic trees. Most of these methods are applied to biomolecular sequences, such as DNA, RNA, or amino-acid sequences. More recently, methods have also been developed to reconstruct phylogenies from data on gene content and gene order within a genome.

Evaluating the performance of phylogenetic reconstruction methods is complicated. Since many phylogenetic reconstruction methods are explicit attempts to solve optimization problems, methods may be compared, for example, with respect to the value of these optimization criteria on both real and synthetic data. However, the biological community has also looked at the performance of these methods with respect to an assumed stochastic model of evolution and has evaluated phylogenetic methods with respect to the topological accuracy of the underlying unrooted trees returned by these methods. Thus, phylogenetic methods are evaluated in two different, yet related, ways. Furthermore, the difficulty in establishing true evolutionary histories for many datasets has led the research community to study these questions largely from results on synthetic, rather than real, data, and to use simulations as the main technique. The design of appropriate simulation studies presents interesting and subtle challenges to the researcher in experimental algorithmics.

Fast implementations of phylogenetic methods are also of potentially tremendous impact, since biologists want to apply these methods to large

datasets (containing hundreds to thousands of taxa)—and some smaller datasets have required nearly one hundred CPU-years of computation on modern machines for acceptable analyses. Thus, algorithm engineering also has an important role to play in this domain.

In this paper, we review the experimental challenges posed by phylogeny reconstruction, in terms both of algorithm engineering and of data generation, collection, and analysis, and present examples from our own experimental research.

8.2 Data for Phylogeny Reconstruction

Phylogenies are most commonly reconstructed using biomolecular sequences (DNA, RNA, or amino acid) for particular genes or non-coding regions of DNA. More recently, "genomic rearrangement" data have been used to infer deep evolutionary histories (i.e., very ancient evolutionary events), as well as to clarify evolutionary relationships in difficult datasets. The use of genome rearrangement data is part of an increased interest in the development of new sources of phylogenetic information, especially those which can be characterized as "rare genomic changes" (see [8.31] for a survey of these approaches). Sequence data and genomic rearrangement data are highly complementary, with different rates of evolution especially in organelles (chloroplasts and mitochondria), so that using both types of data holds potential for improving accuracy in phylogenetic reconstructions.

DNA, RNA, and amino-acid sequences are used for phylogenetic reconstruction. DNA and RNA sequences can be considered simply as strings over a 4-letter alphabet (A, C, T, and G for DNA), while amino-acid sequences can be considered as strings over a 20-letter alphabet (one for each amino-acid). These sequences evolve through events such as substitutions of one nucleotide by another, insertions and deletions of substrings, etc. Typical sources of biomolecular sequence data are individual genes, so that the sequence coding for a given gene is used in each of the relevant taxa. These sequences are then placed in a multiple alignment through the introduction of spaces; each resulting column of the alignment then corresponds to a place in the sequence and changes can be identified as mutations (two sequences have different entries in that column), insertions (a sequence has an entry, but the other has a space), and deletions (the reverse). Computing a good multiple sequence alignment is itself a hard optimization problem, but outside our scope; we direct the interested reader to [8.38] for an introduction to this problem.

Genome rearrangement data indicate how the genes are ordered within the given genomes. Many organellar genomes are composed of a single chromosome and are relatively small, so that every gene within the genome can be identified and their relative ordering inferred (most accurately through whole genome sequencing, but also through gene mapping). Given an ordering of

the genes, we can represent a given genome by an ordering of signed integers. Organellar genomes are thought to evolve via inversions (mechanisms that pick up a segment of a genome and invert it, thus reversing the order of the affected genes), transpositions (mechanisms that pick up a segment of the genome and move it to another position, thus changing the order but not the sign of the affected genes), and inverted transpositions (which are inversions followed by transposition of the inverted segment).

8.2.1 Phylogenetic Reconstruction Methods

For both biomolecular sequences and gene orders, assumptions are made about the mechanism by which these objects evolve. Phylogenetic reconstructions explicitly use assumptions about evolution, but differ in the details of these assumptions. For example, in an analysis of DNA sequences, we may have an explicit model about the evolutionary process; we may know the rates of each type of nucleotide substitution on the true (but unknown) tree. If we assume these rates, then we can seek the tree which is most likely to have generated the given data—the so-called "maximum-likelihood" approach. We can also use these assumptions to infer evolutionary distances between each pair of the given sequences, where the "evolutionary distance" between two sequences is the most likely number of individual changes within the sequence on the path between the two sequences. Both of these approaches have theoretical guarantees, with respect to topological accuracy of the resultant trees, provided that the model is not over-parameterized and that the assumptions about the model are correct. However, maximum-likelihood methods are computationally very intensive, while the second type of methods (called "distance-based" methods) tend to run in polynomial time.

A final class of methods (called "maximum parsimony") does not make any explicit assumption about the model parameters; instead, it seeks a tree with a minimum "number of events". In the context of DNA sequence evolution, these events are nucleotide substitutions, insertions, or deletions, while in the context of gene-order data, they are inversions, transpositions, and inverted transpositions. Maximum parsimony is thus the Steiner Tree Problem for the appropriate space—for instance, the maximum parsimony problem on DNA sequences is the Hamming Distance Steiner Tree problem on strings over a four-letter alphabet. When the input is just the set of taxa (e.g., DNA sequences or gene orders), then the problem is to construct a tree and to label its internal nodes in such a way as to minimize the total number of changes. This problem is NP-hard for both types of data. The point estimation problem, i.e., the scoring of a particular tree topology (in which case the input also includes a specific tree with leaves labelled by the taxa), is solvable in polynomial time for the biomolecular sequence data case, but is NP-hard for gene-order data.

These three types of methods, namely *maximum likelihood*, *distance-based*, and *maximum parsimony*, account for great majority of the methods used by

biologists and their relative performance is passionately argued in the biological literature. One of the major limitations of both maximum parsimony and maximum likelihood techniques (even their heuristic versions, which may not have any performance guarantees) is that they take too long. Even some only moderately large datasets can take years of real analysis (hundreds of CPU years), without resolution [8.29]. By comparison, distance-based methods, including the popular Neighbor-Joining (NJ) method [8.32], are often quite accurate (with respect to topological accuracy, as determined using simulation studies) and are very fast (polynomial-time and fast in practice). While the experimental evidence is not yet definitive, the best distance-based methods appear less accurate than the better heuristics for maximum parsimony and maximum likelihood, at least on large trees with high rates of evolution (see, e.g., [8.13]).

8.3 Algorithmic and Experimental Challenges

8.3.1 Designing for Speed

Because both parsimony- and likelihood-based approaches involve NP-hard optimization problems and because poor approximations may lead to biologically incorrect conclusions, developing efficient exact algorithms is a major concern. The range of data used in current analyses is fortunately limited (e.g., the length of available DNA sequences is bounded, as is the number of genes in a mitochondrial or chloroplast genome). This bounded range is tailor-made for applications of algorithm engineering techniques.

8.3.2 Designing for Accuracy

When exact methods fail to terminate, one needs to use approximations. But it is important to keep in mind that the optimization criterion rarely has direct biological significance, so that deviations from optimal, even by small amounts, may yield results that are grossly different from a biological perspective. Thus the development and evaluation of approximation algorithms must be guided by biological considerations. As discussed earlier, the main criterion by which biologists judge the quality of a reconstruction is its topological accuracy, which is only indirectly related to a parsimony or likelihood criterion. Indeed, the great success of the neighbor-joining heuristic, which has no approximation guarantees for the standard optimization criteria, demonstrates that the biological relevance of results matters more than the traditional algorithmic goal of performance guarantees. Simulation experiments can measure topological accuracy, but designing algorithms to produce topologically correct trees is another matter—the methods most closely oriented toward this goal, the family of quartet-based methods, turns out to produce generally much poorer trees than the much simpler neighbor-joining algorithm.

8.3.3 Performance Evaluation

Phylogenetic reconstruction methods are evaluated according to three basic types of criteria: *statistical performance*, which addresses the accuracy of the method under a specified stochastic model of evolution; *computational performance*, which addresses the computational requirements of the method; and *data requirements*, which addresses the requirements, in terms of quality and quantity, placed on the input data. Accuracy in a phylogenetic reconstruction method is determined primarily by comparing the unrooted leaf-labelled tree obtained by the method to the "true" tree. Since the true tree is usually unknown, accuracy is addressed either theoretically, with reference to a fixed but unknown tree in some model of evolution, or through simulation studies. A method is said to be *accurate* if the tree obtained is exactly equal to the unrooted version of the model (or true) tree; degrees of accuracy are quantified typically by the percentage of the edges of the true tree that occur in the estimated tree. A method is said to be *statistically consistent* with respect to a specific model of evolution if it is guaranteed to recover the true tree with probability going to 1 as the amount of data (e.g., sequence length) goes to infinity. The latter property is not as good as it may sound: nature provides us with finite data only—for instance, DNA sequences cannot be of arbitrary length, much less gene orders; thus the rate of convergence is crucial and needs to be evaluated experimentally as well as bounded theoretically (see [8.35] for such an evaluation and [8.37] for a theoretical approach). Data requirements therefore loom large—and indeed may prove more detrimental than computational requirements, since we can always run the program longer.

Because the evolutionary models that biologists favor are parameter-rich, experimental assessment of performance (whether accuracy, convergence rate, or running time) is a daunting task: choosing how to vary the parameters while keeping the total computation down is a difficult tradeoff.

8.4 An Algorithm Engineering Example: Solving the Breakpoint Phylogeny

Blanchette *et al.* [8.6] developed an approach, which they called *breakpoint phylogeny*, for reconstructing phylogenies from gene order data. Their approach is limited to the special case in which the genomes all have the same set of genes and each gene appears once. This special case is of interest to biologists, who hypothesize that inversions (which can only affect gene order, but not gene content) are the main evolutionary mechanism for a range of genomes or chromosomes (chloroplast, mitochondria, human X chromosome, etc.). Simulation studies we conducted suggested that this approach works well for certain datasets (i.e., it obtains trees that are close to the model tree), but that the implementation developed by Sankoff and Blanchette, the

```
For each tree topology do:
  Initially label all internal nodes with gene orders
  Repeat
    For each internal node v, with neighbors A, B, and C, do
        Solve the MPB on A, B, C to yield label m
        If relabelling v with m improves the score of T, then do it
  until no internal node can be relabelled
```

Fig. 8.2. `BPAnalysis`

`BPAnalysis` software [8.33], is too slow to be used on anything other than small datasets with a few genes [8.9, 8.10].

8.4.1 Breakpoint Analysis: Details

When each genome has the same set of genes and each gene appears exactly once, a genome can be described by an ordering (circular or linear) of these genes, each gene given with an orientation that is either positive (g_i) or negative ($-g_i$). Given two genomes G and G' on the same set of genes, a *breakpoint* in G is defined as an ordered pair of genes, (g_i, g_j), such that g_i and g_j appear consecutively in that order in G, but neither (g_i, g_j) nor $(-g_j, -g_i)$ appears consecutively in that order in G'. The breakpoint distance between two genomes is the number of breakpoints between that pair of genomes. The breakpoint score of a tree in which each node is labelled by a signed ordering of genes is then the sum of the breakpoint distances along the edges of the tree.

Given three genomes, we define their *median* to be a fourth genome that minimizes the sum of the breakpoint distances between it and the other three. The *Median Problem for Breakpoints* (MPB) is to construct such a median and is NP-hard [8.27]. Sankoff and Blanchette developed a reduction from MPB to the Travelling Salesman Problem (TSP), perhaps the most studied of all optimization problems [8.15]. Their reduction produces an undirected instance of the TSP from the directed instance of MPB by the standard technique of representing each gene by a pair of cities connected by an edge that must be included in any solution.

`BPAnalysis` (see Figure 8.2) is the method developed by Blanchette and Sankoff to solve the breakpoint phylogeny. Within a framework that enumerates all trees, it uses an iterative heuristic to label the internal nodes with signed gene orders. This procedure is computationally very intensive. The outer loop enumerates all $(2n-5)!!$ leaf-labelled trees on n leaves, an exponentially large value.[1] The inner loop runs an unknown number of iterations (until convergence), with each iteration solving an instance of the TSP (with a number of cities equal to twice the number of genes) at each internal node.

[1] The double factorial is a factorial with a step of 2, so we have $(2n-5)!! = (2n-5) \cdot (2n-7) \cdot \ldots \cdot 3$.

The computational complexity of the entire algorithm is thus exponential in *each* of the number of genomes and the number of genes, with significant coefficients. The procedure nevertheless remains a heuristic: even though all trees are examined and each MPB problem solved exactly, the tree-labeling phase does not ensure optimality unless the tree has only three leaves.

8.4.2 Re-Engineering BPAnalysis for Speed

Profiling. Algorithmic engineering suggests a refinement cycle in which the behavior of the current implementation is studied in order to identify problem areas which can include excessive resource consumption or poor results. We used extensive profiling and testing throughout our development cycle, which allowed us to identify and eliminate a number of such problems. For instance, converting the MPB into a TSP instance dominates the running time whenever the TSP instances are not too hard to solve. Thus we lavished much attention on that routine, down to the level of hand-unrolling loops to avoid modulo computations and allowing reuse of intermediate expressions; we cut the running time of that routine down by a factor of at least six—and thereby nearly tripled the speed of the overall code. We lavished equal attention on distance computations and on the computation of the lower bound, with similar results. Constant profiling is the key to such an approach, because the identity of the principal "culprits" in time consumption changes after each improvement, so that attention must shift to different parts of the code during the process—including revisiting already improved code for further improvements. These steps provided a speed-up by one order of magnitude on the Campanulaceae dataset.

Cache Awareness. The original `BPAnalysis` is written in C++ and uses a space-intensive full distance matrix, as well as many other data structures. It has a significant memory footprint (over 60MB when running on the Campanulaceae dataset) and poor locality (a working set size of about 12MB). Our implementation has a tiny memory footprint (1.8MB on the Campanulaceae dataset) and good locality (all of our storage is in arrays preallocated in the main routine and retained and reused throughout the computation), which enables it to run almost completely in cache (the working set size is 600KB). Cache locality can be improved by returning to a FORTRAN-style of programming, in which storage is static, in which records (structures/classes) are avoided in favor of separate arrays, in which simple iterative loops that traverse an array linearly are preferred over pointer dereferencing, in which code is replicated to process each array separately, etc. (This style of programming is not always easy to reconcile with the currently favored object-oriented style; fortunately, compiler support for this type of code optimization is slowly developing—as of January 2002, for instance, at least one commercial compiler could optimize storage access by breaking an array of record into multiple arrays. We found it easier to code in C, simply because of

the much greater transparency of the language.) While we cannot measure exactly how much we gain from this approach, studies of cache-aware algorithms [8.1, 8.11, 8.18, 8.19, 8.20, 8.39] indicate that the gain is likely to be substantial—factors of anywhere from 2 to 40 have been reported. New memory hierarchies show differences in speed between cache and main memory that exceed two orders of magnitude.

Low-Level Algorithmic Changes. Unless the original implementation is poor (which was not the case with `BPAnalysis`), profiling and cache-aware programming will rarely provide more than two orders of magnitude in speed-up. Further gains can often be obtained by low-level improvement in the algorithmic details. In our phylogenetic software, we made two such improvements. The basic algorithm scores every single tree, which is clearly very wasteful; we used a simple lower bound, computable in linear time, to enable us to eliminate a tree without scoring it. On the Campanulaceae dataset, this bounding eliminates over 99.95% of the trees without scoring them, resulting in a 100-fold speed-up. The TSP solver we wrote is at heart the same basic include/exclude search as in `BPAnalysis`, but we took advantage of the nature of the instances created by the reduction to make the solver much more efficient, resulting in a speed-up by a factor of 5–10. These improvements all spring from a careful examination of exactly what information is readily available or easily computable at each stage and from a deliberate effort to make use of all such information.

A High-Performance Implementation. Our implementation, *GRAPPA*[2], incorporates all of the refinements mentioned above, plus others specifically made to enable the code to run efficiently in parallel (see [8.23, 8.24, 8.26] for details). Because the basic algorithm enumerates and independently scores every tree, it presents obvious parallelism: we can have each processor handle a subset of the trees. In order to do so efficiently, we need to impose a linear ordering on the set of all possible trees and devise a generator that can start at an arbitrary point along this ordering. Because the number of trees is so large, an arbitrary tree index would require unbounded-precision integers, considerably slowing down tree generation. Our solution was to design a tree generator that starts with tree index k and generates trees with indices $\{k + cn \mid n \in \mathcal{N}\}$, where k and c are regular integers, all without using unbounded-precision arithmetic. Such a generator allows us to sample tree space (a very useful feature in research) and, more importantly, allows us to use a cluster of c processors, where processor i, $0 \leq i \leq c-1$, generates and scores trees with indices $\{i + cn \mid n \in \mathcal{N}\}$. We ran *GRAPPA* on the 512-processor Alliance cluster *Los Lobos* at the University of New Mexico and obtained a 512-fold speed-up. When combined with the nearly 200,000-fold speedup obtained through algorithm engineering, our run on the Campanulaceae dataset demonstrated a *one hundred million-fold* speed-up over the

[2] Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms.

original implementation [8.26] (a first speedup of one million was reported in [8.2]).

8.4.3 A Partial Assessment

Clearly, generating every single tree is a self-defeating approach: even our huge 10^8-fold speedup allowed us to move from 10 taxa to just 16 taxa—and 20 or more taxa remain forever out of reach of this computational approach. A real search strategy should reduce the cost of computation by an enormous factor. Yet this exercise in algorithm engineering already produced significant results in both biology and computer science: the analysis of the *Campanulaceae* dataset conformed to the expectations of the biologists and thus reinforced the conjecture that gene-order data carries significant information about evolution (and, incidentally, that inversion-driven rearrangements are indeed the main mechanism for such evolution), while the improved understanding of inversion distance computations gained through the implementation enabled us to design the first true linear-time algorithm for this purpose. In turn, the availability of a fast implementation for inversion distance computations and inversion-based phylogenies has spurred renewed interest in the inversion median problem [8.7, 8.34] and other related problems.

8.5 An Experimental Algorithmics Example: Quartet-Based Methods for DNA Data

8.5.1 Quartet-Based Methods

A *quartet tree* is an unrooted binary tree on four taxa. A quartet tree thus induces a unique bipartition of the four taxa and can be denoted by that bipartition. If the taxa are $\{a, b, c, d\}$, we can use $\{ab|cd\}$ to denote the quartet tree that pairs a with b and c with d (see Figure 8.3). A quartet tree $\{ab|cd\}$ *agrees* with a tree T if all four of its taxa are leaves of T and the path from a to b in T does not intersect the path from c to d in T. Equivalently, $\{ab|cd\}$ agrees with a tree if the subtree induced in T by the four taxa is the quartet tree itself. The quartet tree $\{ab|cd\}$ *is an error* with respect to the tree T if it does not agree with T. If $Q(T)$ denotes the set of all quartet trees that agree with T, then T is uniquely characterized by $Q(T)$ and can be reconstructed from $Q(T)$ in polynomial time [8.12].

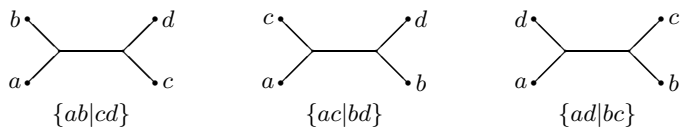


Fig. 8.3. The three possible quartet trees on four taxa $\{a, b, c, d\}$ and their bipartition encodings

Quartet-based methods operate in two phases. In the first phase they construct a set Q of quartet trees on the different sets of four taxa; in the second phase, they combine these quartet trees into a tree on the entire set of taxa. In practice, the input data are not of sufficient quality to ensure that all quartet trees are accurately inferred, so that quartet methods have to find ways of handling incorrect quartet trees. With the exception of Quartet Puzzling, all quartet methods we examine provide guarantees about the edges of the true tree that they reconstruct. These guarantees are expressed in terms of "quartet errors around an edge," a concept we now define.

Consider an edge e in the true tree T; its removal defines the bipartition $A|B$ on the leaves of S. Consider those sets of four leaves $\{a, a', b, b'\}$ with $\{a, a'\} \subseteq A$ and $\{b, b'\} \subseteq B$. A quartet tree t is said to be an "error around e" if we have $t = \{ab|a'b'\}$ or $t = \{ab'|a'b\}$. Similarly, if T' is a proposed tree and Q is a set of quartet trees, then $t \in Q$ is an error around edge $e \in E(T')$ if $t = \{ab|a'b'\}$ or $t = \{ab'|a'b\}$, while e defines the bipartition $A|B$.

Two of the methods we study, the Q^* method (also known as the Buneman method) and the Quartet-Cleaning methods, can be described in terms of an explicit bound on the number of quartet errors around the edges they reconstruct. The Q^* method [8.4] seeks the *maximally resolved* tree T' obeying $Q(T') \subseteq Q$; therefore, there are *no quartet errors* around any edge in the tree T'. *Quartet-Cleaning (QC)* methods [8.3, 8.5, 8.14] have explicit bounds on the number of quartet errors around each reconstructed edge e. These error bounds have the form $m\sqrt{q_e}$, where q_e is the number of quartet trees around edge e and m is a small constant. Thus, the Q^* method is a cleaning method with $m = 0$. The *global cleaning* method sets $m = 1$ and the *local cleaning* method sets $m = \frac{1}{2}$; these methods are guaranteed to recover every edge of the true tree for which Q contains a small enough number of quartet errors. The *hypercleaning* method allows m to be an arbitrary integer and thus has the potential to recover more edges, at the cost of a high running time (proportional to $n^7 \cdot m^{4m+2}$), so that it is impractical for m larger than 5.

The final quartet-based method we examined is the best known and the most frequently used by biologists [8.22, 8.30, 8.17]: the *Quartet-Puzzling (QP)* method [8.36]. This heuristic computes quartet trees using maximum likelihood (ML) and then uses a greedy strategy to construct a tree on which many input quartets are in agreement. QP uses an arbitrary ordering of taxa, constructs the optimal quartet tree on the first four, then inserts each successive taxon in turn, attaching the new leaf to an edge of the current tree so as to optimize a quartet-based score. Because the input ordering of taxa is pertinent, QP uses a large number of random input orderings and computes the *majority consensus* of all trees found. (The majority consensus is the tree that contains all bipartitions that appear in more than half of the trees in the set and is commonly used by biologists.)

8.5.2 Experimental Design

We used Jukes-Cantor model trees with varying numbers of taxa and rates of evolution to generate a large number of synthetic datasets of varying lengths. (The Jukes-Cantor model [8.16] is the simplest of the various evolutionary models, with just one parameter.) For each dataset generated, we computed the neighbor-joining (NJ) and QP trees on the entire dataset and two sets of quartet trees, one based upon ML, Q_{ML}, and one based upon NJ, Q_{NJ}. We then applied various cleaning methods to each of the sets Q_{ML} and Q_{NJ}. We compared quartet trees of Q_{ML}, of Q_{NJ}, and of the reconstructed trees, as well as the reconstructed trees themselves, against the model tree for accuracy.

We randomly generated model tree topologies from the uniform distribution on binary leaf-labelled trees. For each edge of each tree topology, we generated a random number (from the uniform distribution) between 1 and 1000 and used that number as the initial "length" of the edge. We then scaled each such "base" model tree by a multiplicative factor, ranging from 10^{-7} to 10^{-3}. This process produces Jukes-Cantor trees with edge lengths (λ_e for edge e) ranging from a minimum of 10^{-7} to a maximum of 1. The edge length denotes the probability that a particular character in the sequence at the base of the edge will be affected by an evolutionary event along the edge; thus the expected number of changes affecting the sequence at the base of the edge is the product of the edge length by the sequence length. In the following we write $\overline{\lambda_e}$ to denote the average edge length in a collection of trees—which is just 500 times the scaling factor. We generated random DNA sequences for the root and used the program `Seq-Gen` [8.28] to evolve these sequences down the tree under the Jukes-Cantor model of evolution, thus producing sets of sequences at the leaves, our synthetic datasets.

Because the number of distinct unrooted, leaf-labelled trees on n leaves is $(2n-5)!!$ and because our input space is further expanded by the choice of evolutionary rates, it is not possible to take a fair sample of the entire input space. In order to obtain statistically robust results, we followed the advice of McGeoch [8.21] and Moret [8.25] and used a number of *runs*, each composed of a number of *trials* (a trial is a single comparison), computed the mean outcome for each run, and studied the mean and standard deviation over the runs of these events.

A critical parameter of our study, one that has not been explored in most prior studies, is the number of input taxa. Previous experimental studies have often been limited to a small number of taxa due to computational problems. However, to resolve phylogenetic trees of interest to biologists, algorithms must scale reasonably, both in terms of topological accuracy and running time, to problems of the size that biologists typically study (20–200 taxa), as well as those they would like to address (a few hundred to several thousand taxa).

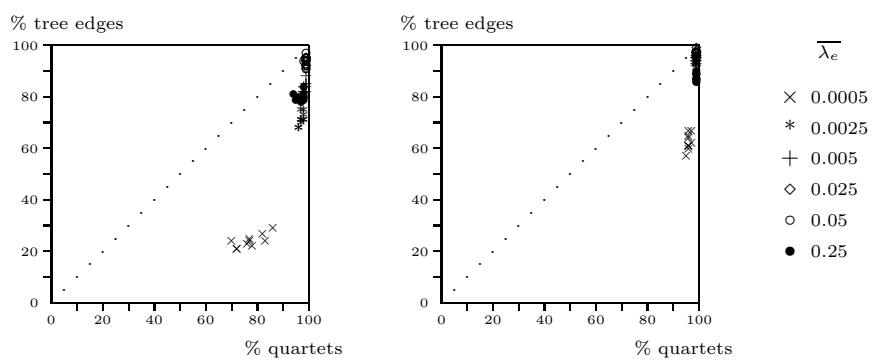
We ran our test suite for 5, 10, 20, 40, and selected sets of 80 taxa. Our tests included a selection of eight expected evolutionary rates, from $5 \times$

10^{-5} to 5×10^{-1} per tree edge. For each evolutionary rate and problem size, we generated a total of 100 topologies, grouped into ten runs of ten trials. All tests were conducted for four sequence lengths: 500, 2,000, 8,000, and 32,000. We note that sequence lengths above 1,000 are considered long and those above 5,000 extremely long; thus our study explores longer sequence lengths than are usually encountered in practice. In all, our study used 16,000 datasets and required many months of computation on two medium-sized clusters.

Our focus was the accuracy of solutions generated by the various tree reconstruction methods. To assess topological accuracy, we measured the number of true positives (edges of the true tree that appear in the reconstructed tree). For cleaning methods, we measured these values before and after cleaning. For each run of ten trials, we retained only the mean values. Our results are composed of the means for each set of ten runs.

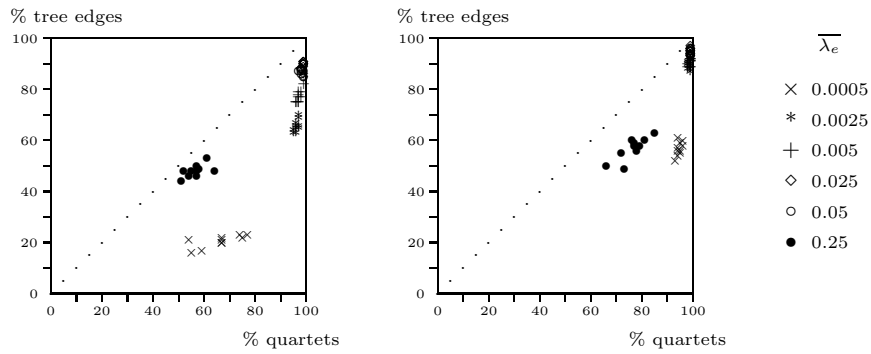
8.5.3 Some Experimental Results

We provide only a few illustrative results from our study [8.35]. Because our focus was accuracy, we wanted to find out whether the goal of minimizing quartet errors would correlate closely with the true goal of maximizing topological accuracy. Our results showed convincingly that topological accuracy is a more demanding criterion than quartet accuracy and should therefore be used to assess performance of phylogenetic reconstruction methods; typical results are shown in Figures 8.4 and 8.5. Both NJ and QP can return trees with only 20% of the edges correct from a set of quartet trees that is 80% correct. Worse yet, both methods, except when the percentage of correct quartet trees is close to 100%, can return fewer than 80% of the true tree edges (in



(a) seq. length 500 (b) seq. length 2000

Fig. 8.4. Percent of true tree edges recovered by global NJ for various $\overline{\lambda_e}$ as a function of the percentage of correct induced quartet trees for 40 taxa and two sequence lengths



(a) seq. length 500 (b) seq. length 2000

Fig. 8.5. Percent of true tree edges recovered by QP for various $\overline{\lambda_e}$ as a function of the percentage of correct induced quartet trees for 40 taxa and two sequence lengths

the case of QP, some such trees had only 60% of the true tree edges). Because failure to obtain at least 90 or 95% of the edges can be unacceptable to systematists, quartet-based measures of accuracy are not acceptable surrogates for true tree edges.

Theory predicts that the accuracy of methods will degrade as the number of taxa increases while sequence length and average edge length (the expected number of changes for a random site on each edge) are held fixed. Figure 8.6 shows the topological accuracy achieved by all six methods as a function of the number of taxa for a sequence length of 500 and for three different average edge lengths. All methods decrease in accuracy as the number of taxa increases, even though both NJ and QP show an initial increase (particularly for lower evolutionary rates). QC provides a distinct improvement over the Q^* method, whether the quartet trees are computed using ML or local NJ. QCML and QCNJ are very close in performance, although QCNJ slightly outperforms QCML; similarly Q^*NJ slightly outperforms Q^*ML. Of the five quartet methods, QP is the best throughout the range of parameters studied, but NJ completely dominates it.

8.6 Observations and Conclusions

Our two examples illustrate two different facets of computational phylogenetics: the first shows that algorithmic engineering can turn what appears to be strictly a proof of concept into a usable tool, while the second demonstrates the scale required in good experimentation when assessing the behavior of reconstruction algorithms. It is worth stressing that the goal of a biologist is to analyze a specific dataset—so that the biologist will not mind running a cluster on the problem for as long as necessary (weeks or months), since

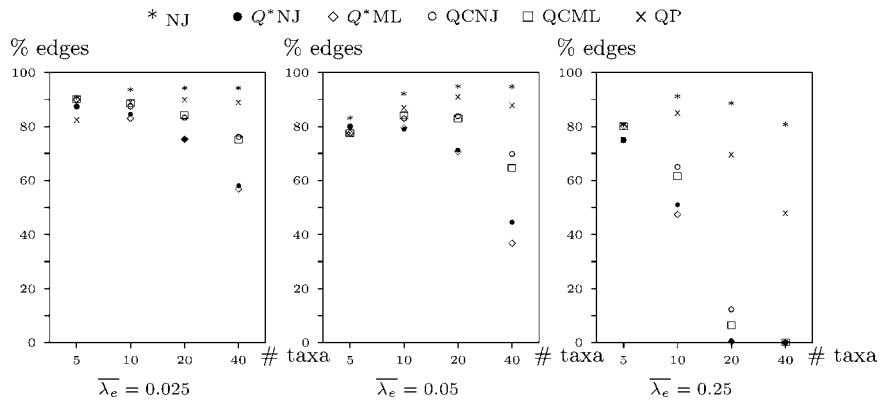


Fig. 8.6. Number of taxa *vs.* percentage of edges correct for sequence length 500 and various $\overline{\lambda_e}$

just one instance must be solved. In contrast, the role of the algorithm designer or engineer is to document the practical performance of an algorithm, which requires many runs on many different sizes and types of data. In consequence, an algorithm that a biologist finds acceptable computationally may be judged completely inadequate by an algorithm engineer—a discrepancy that can only be resolved through a close collaboration between the biologist and the algorithm engineer.

A boon to the experimentalist in the area is the availability of real datasets: academic biologists are generally very free with their data and only too pleased to have an algorithm designer help them answer their questions. We have a national database that stores most known DNA sequences (GenBank) and government laboratories that sequence organellar genomes, providing vast amounts of challenging problems to the algorithms community. On the other hand, the absence of consensus on a model of evolution makes it difficult to obtain definitive results. Current models appear to be brittle, in the sense that small deviations from optimality may cause significant degradation of the quality of the solution as perceived by a biologist; the optimality criteria need to be refined to remedy this problem. Again, working with a research biologist is crucial, as the biologist can sift through the changes in model parameters and the output produced under each model and prepare an analysis and, if necessary, a new model.

On the algorithm engineering side, the area needs a large effort in code production, code that is made freely available to biologists everywhere, in order to replace poor existing codes, implement new ideas, and provide the most efficient tools to the biologists. Perhaps more than in any other area, there is an opportunity in computational biology, in particular in computational phylogenetics, for algorithm designers and engineers to have a profound impact on the course of scientific research.

Acknowledgments

This work was supported in part by NSF grants CCR 94-57800 (Warnow), ACI 00-81404 (Moret), DEB 01-20709 (Moret and Warnow), EIA 01-13095 (Moret), EIA 01-13654 (Warnow), EIA 01-21377 (Moret), and EIA 01-21680 (Warnow), and by the David and Lucile Packard Foundation (Warnow).

References

8.1 L. Arge, J. Chase, J. S. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. In *Proceedings of the 4th Workshop on Algorithm Engineering (WAE'00)*. Springer Lecture Notes in Computer Science 1982, 2000.

8.2 D. A. Bader and B. M. E. Moret. GRAPPA runs in record time. *HPC Wire*, 9(47), 2000.

8.3 V. Berry, D. Bryant, T. Jiang, P. Kearney, M. Li, T. Wareham, and H. Zhang. A practical algorithm for recovering the best supported edges of an evolutionary tree. In *Proceedings of the 11th ACM/SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 287–296, 2000.

8.4 V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240(2):271–298, 2000.

8.5 V. Berry, T. Jiang, P. Kearney, M. Li, and T. Wareham. Quartet cleaning: improved algorithms and simulations. In *Proceedings of the 7th European Symposium on Algorithms (ESA'99)*. Springer Lecture Notes in Computer Science 1643, pages 313–324, 1999.

8.6 M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In S. Miyano and T. Takagi, editors, *Genome Informatics 1997*, pages 25–34. Univ. Academy Press, Tokyo, 1997.

8.7 A. Caprara. On the practical solution of the reversal median problem. In *Proceedings of the 1st Workshop on Algorithms for Bioinformatics (WABI'01)*. Springer Lecture Notes in Computer Science 2149, pages 238–251, 2001.

8.8 J. I. Cohen. Epstein-barr virus infection. *New England Journal of Medicine*, 343(7):481–492, 2000.

8.9 M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L.-S. Wang, T. Warnow, and S. K. Wyman. An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae. In D. Sankoff and J. Nadeau, editors, *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families*, pages 99–121. Kluwer, 2000.

8.10 M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L. Wang, T. Warnow, and S. K. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00)*, pages 104–115, 2000.

8.11 N. Eiron, M. Rodeh, and I. Stewarts. Matrix multiplication: a case study of enhanced data cache utilization. *ACM Journal of Experimental Algorithmics*, 4(3), 1999. Online at `www.jea.acm.org/1999/EironMatrix/`.

8.12 P. Erdős, M. A. Steel, L. A. Székely, and T. Warnow. A few logs suffice to build (almost) all trees I. *Random Structures and Algorithms*, 14:153–184, 1997.

8.13 D. Huson, S. Nettles, K. Rice, T. Warnow, and S. Yooseph. Hybrid tree reconstruction methods. *ACM Journal of Experimental Algorithmics*, 4(5), 1999. Online at `www.jea.acm.org/1999/HusonHybrid/`.

8.14 T. Jiang, P. E. Kearney, and M. Li. A polynomial-time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM Journal on Computing*. To appear.

8.15 D. S. Johnson and L. A. McGeoch. The traveling salesman problem: a case study. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley, 1997.

8.16 T. H. Jukes and C. Cantor. *Mammalian Protein Metabolism*. Academic Press, 1969.

8.17 P. J. Keeling, M. A. Luker, and J. D. Palmer. Evidence from beta-tubulin phylogeny that microsporidia evolved from within the Fungi. *Molecular Biology and Evolution*, 17:23–31, 2000.

8.18 R. Ladner, J. D. Fix, and A. LaMarca. The cache performance of traversals and random accesses. In *Proceedings of the 10th ACM/SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 613–622, 1999.

8.19 A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1(4), 1996. Online at `www.jea.acm.org/1996/LaMarcaInfluence/`.

8.20 A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th ACM/SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 370–379, 1997.

8.21 C. C. McGeoch. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24:195–212, 1992.

8.22 B. Mishof, C. L. Anderson, and H. Hadrys. A phylogeny of the damselfly genus *Calopteryx* (Odonata) using mitochondrial 16s rDNA markers. *Molecular Phylogeny Evolution*, 15:5–14, 2000.

8.23 B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. In *Proceedings of the 2001 International Conference on Computational Science (ICCS'01)*. Springer Lecture Notes in Computer Science 2073–2074, 2001.

8.24 B. M. E. Moret, S. K. Wyman, D. A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium Biocomputing (PSB'01)*. World Scientific, pages 583–594, 2001.

8.25 B. M. E. Moret and H. D. Shapiro. Algorithms and experiments: the new (and old) methodology. *Journal on Universal Computer Science*, 7(5):434–446, 2001.

8.26 B. M. E. Moret, J. Tang, L.-S. Wang, and T. Warnow. Steps toward accurate reconstruction of phylogenies from gene-order data. *Journal on Computer and System Sciences*. To appear.

8.27 I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. *Electronic Colloqium on Computational Complexity*, 71, 1998.

8.28 A. Rambaut and N. C. Grassly. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Computational Applications in Biosciences*, 13:235–238, 1997.

8.29 K. Rice, M. Donoghue, and R. Olmstead. Analyzing large datasets: rbcl500 revisited. *System Biology*, 46:554–562, 1997.

8.30 F. Rodrigues-Trelles, L. Alarcon, and A. Fontdevila. Molecular evolution and phylogeny of the *buzzatii* complex (*D. repleta* group): a maximum likelihood approach. *Molecular Biology Evolution*, 17:1112–1122, 2000.

8.31 A. Rokas and P. W. H. Holland. Rare genomic changes as a tool for phylogenetics. *Trends in Ecology and Evolution*, 15:454–459, 2000.
8.32 N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology Evolotion*, 4:406–425, 1987.
8.33 D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal on Computational Biology*, 5:555–570, 1998.
8.34 A. C. Siepel and B. M. E. Moret. Finding an optimal inversion median: experimental results. In *Proceedings of the 1st Workshop on Algorithms for Bioinformatics (WABI'01)*. Springer Lecture Notes in Computer Science 2149, pages 189–203, 2001.
8.35 K. St. John, T. Warnow, B. M. E. Moret, and L. Vawter. Performance study of phylogenetic methods: (unweighted) quartet methods and neighbor-joining. In *Proceedings of the 12th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 196–205, 2001.
8.36 K. Strimmer and A. von Haeseler. Quartet puzzling: a maximum likelihood method for reconstructing tree topologies. *Molecular Biology Evolution*, 13:964–969, 1996.
8.37 T. Warnow, B. M. E. Moret, and K. St. John. Absolute phylogeny: true trees from short sequences. In *Proceedings of the 12th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 186–195, 2001.
8.38 M. S. Waterman. *Introduction to Computational Biology: Sequences, Maps and Genomes*. Chapman Hall, 1995.
8.39 L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics*, 5(3), 2000. Online at `www.jea.acm.org/2000/XiaoMemory/`.

9. Presenting Data from Experiments in Algorithmics

Peter Sanders*

Max-Planck-Institut für Informatik, Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Summary.

Algorithmic experiments yield large amounts of data that depends on many parameters. This paper collects a number of rules for presenting this data in concise, meaningful, understandable graphs that have sufficiently high quality to be printed in scientific journals. The focus is on common sense rules that are frequently useful and can be easily implemented using tools such as gnuplot[1].

9.1 Introduction

A paper in experimental algorithmics will often start by describing the problem and the experimental setup. Then a substantial part will be devoted to presenting the results together with their interpretation. Consequently, compiling the measured data into graphs is a central part of writing such a paper. This problem is often rather difficult because several competing factors are involved. First, the measurements can depend on many parameters: problem size and other quantities describing the problem instance; variables like number of processors, available memory describing the machine configuration used; and the algorithm variant together with tuning parameters such as the cooling rate in a simulated annealing algorithm.

Furthermore, many quantities can be measured such as solution quality, execution time, memory consumption and other more abstract complexity measures such as the number of comparisons performed by a sorting algorithm. Mathematically speaking, we sample function values of a mapping $f : A \to B$ where the domain A can be high-dimensional. We hope to uncover properties of f from the measurements, e.g., an estimate of the time complexity of an algorithm as a function of the input size. Measurement errors may additionally complicate this task.

As a consequence of the the multitude of parameters, a meaningful experimental setup will often produce large amounts of data and still cover only a tiny fraction of the possible measurements. This data has to be presented

* This work was partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[1] www.gnuplot.org. The source codes of the examples in this paper can be found under http://www.mpi-sb.mpg.de/~sanders/gnuplot/

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 181–196, 2002.
© Springer-Verlag Berlin Heidelberg 2002

in a way that clearly demonstrates the observed properties. The most important presentation usually takes place in conference proceedings or scientific journals where limited space and format restriction further complicate the task.

This paper collects rules that have proven to be useful in designing good graphs. Although the examples are drawn from the work of the author, this paper owes a lot to discussions with colleagues and detailed feedback from several referees. Sections 9.3–9.7 explains the rules. The stress is on Section 9.4 where two-dimensional figures are discussed in detail.

Instead of an abstract conclusion, Section 9.8 collects all the rules in a check list that can possibly be used when looking for teaching and as a source of ideas for improving graphs.

Related Work

A number of papers on the methodology of experimental algorithmics have come out recently [9.10, 9.8, 9.9, 9.6]. In particular, [9.6] explains some of the rules presented here.

There are also entire books on presenting data graphically [9.5, 9.4, 9.17]. The role of the present paper is to formulate domain specific rules, to adapt and specialize more abstract rules and to summarize less important rules. For example, the main emphasis of the above books is on approaches to visualize a limited set of data items in ways which discern structure. Tufte even reports that 75 % of the graphics found in newspapers and magazines are time series — a species of graphs rather rare in algorithmics, where we often face a different situation. We have instance generators which provide us with an unlimited supply of examples and we have control over many parameters. Furthermore, we can repeat experiments as often as we want and hence can often reduce measurement errors to quite small values. The difficulty is now to select the right measurements and display a large amount of data in a compact way.

Another active area of research is the visualization of large amounts of data using three-dimensional, colored animations. Here we limit ourselves to simple graphs suited for black-and-white printing that can be produced with off-the-shelf tools like gnuplot. This paper should not be regarded as a research paper but as a collection of "folklore" rules.

9.2 The Process

In a simplified model of experimental algorithmics a paper might be written using a "waterfall model". The experimental design is followed by a description of the measurement which is in turn followed by an interpretation. In reality, there are numerous feedbacks involved and some might even remain

visible in a presentation. After an algorithm has been implemented, one typically builds a simple yet flexible tool that allows many kinds of measurements. After some explorative measurements the researcher gets a basic idea of interesting parameter settings. Hypotheses are formed which are tested using more extensive measurements using particular parameter ranges. This phase is the scientifically most productive phase and often leads to new insights which lead to algorithmic changes which influence the entire setup.

It should be noted that most algorithmic problems are so complex that one cannot expect to arrive at an ultimate set of measurements that answers all conceivable questions. Rather, one is constantly facing a list of interesting open questions that require new measurements. The process of selecting the measurements that are actually performed is driven by risk and opportunity: The researcher will usually have a set of hypotheses that have some support from measurements but more measurements might be important to confirm them. For example, the hypothesis might be "my algorithm is better than all the others" then a big risk might be that a promising other algorithm or important classes of problem instances have not been tried yet. A small risk might be that a tuning parameter has so far been set in an ad hoc fashion where it is clear that it can only improve a precomputation phase that takes 20 % of the execution time.

An opportunity might be a new idea of the authors' that an algorithm might be useful for a new application where it was not originally designed for. In that case, one might consider to include problem instances from the new application into the measurements.

At some point, a group of researchers decides to cast the current state of results into a paper. The explorative phase is then stopped for a while. To make the presentation concise and convincing, alternative ways to display the data are designed that are compact enough to meet space restrictions and make the conclusions evident. This might also require additional measurements giving additional support to the hypotheses studied.

9.3 Tables

Tables are easier to produce than graphs and perhaps this advantage causes that they are often overused. Tables are more difficult to interpret and too large for large data sets. A more detailed explanation why tables are often a bad idea has been given by McGeoch and Moret [9.9]. Nevertheless, tables have their place. Tufte [9.17] gives the rule of thumb that "tables usually outperform a graph for small data sets of 20 numbers or less". Tables give very accurate values which make it easier to check whether some experiments can be reproduced. Furthermore, one sometimes wants to present some quantities, e.g., solution quality, as a function of problem instances which cannot be meaningfully arranged on the axis of a graph. In that case, a graph or bar chart may look nicer but does not add utility compared to a more accurate

and compact table. Often a paper will contain small tables with particularly important results and graphs giving results in an abstract yet less accurate way. Furthermore, there may be an appendix or a link to a web page containing larger tables for more detailed documentation of the results.

9.4 Two-Dimensional Figures

As our standard example we will use the case that execution time should be displayed as a function of input size. The same rules will usually apply for many other types of variables. Sometimes we mention special examples which should be displayed differently.

9.4.1 The x-Axis

The first question one can ask oneself is what unit one chooses for the x-axis. For example, assume we want to display the time it takes to broadcast a message of length k in some network where transmitting k' bytes of data from one processor to another takes time $t_0 + k'$. Then it makes sense to plot the execution time as a function of k/t_0 because for many implementations, the shape of the curve will then become independent of t_0. More generally, by choosing an appropriate unit, we can sometimes get rid of one degree of freedom. Figure 9.1 gives an example.

The variable defining the x-axis can often vary over many orders of magnitude. Therefore one should always consider whether a logarithmic scale is appropriate for the x-axis. This is an accepted way to give a general idea of a function over a wide range of values. One will then choose measurement values such that they are about evenly spaced on the x-axis, e.g., powers of two or powers of $\sqrt{2}$. Figures 9.3, 9.5, and 9.6 all use powers of two. In this case, one should also choose tic marks which are powers of two and not powers of ten. Figures 9.1 and 9.4 use the "default" base ten because there is no choice of input sizes involved here.

Sometimes it is appropriate to give more measurements for small x-values because they are easily obtained and particularly important. Conversely, it is not a good idea to measures using constant offsets ($x \in \{x_0 + i\Delta : 0 \leq i < i_{\max}\}$) as if one had a linear scale and then to display the values on a logarithmic scale. This looks awkward because points are crowded for large values. Often there will be too few values for small x and one nevertheless wastes a lot of measurement time for large inputs.

A plain linear scale is adequate if the interesting range of x-values is relatively small, for example if the x-axis is the number of processors used and one measures on a small machine with only 8 processors. A linear scale is also good if one wants to point out periodic behavior, for example if one wants to demonstrate that slow-down due to cache conflicts get very large

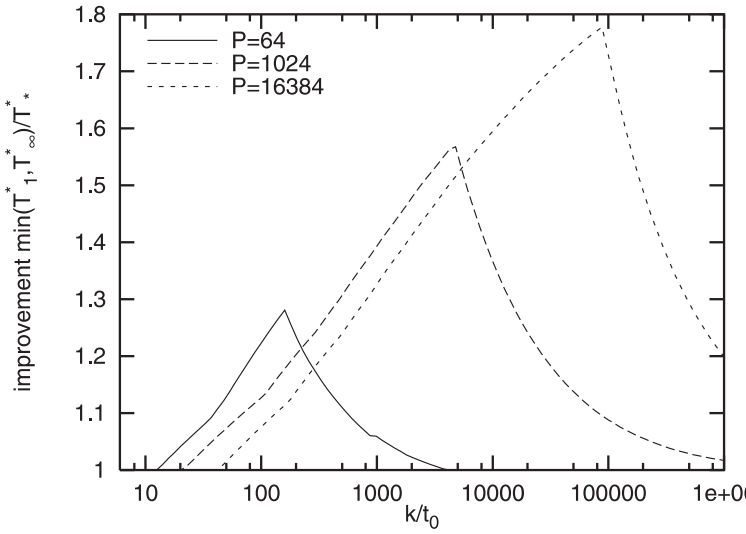


Fig. 9.1. Improvement of the fractional tree broadcasting algorithm [9.15] over the best of pipelined binary tree and sequential pipeline algorithm as a function of message transmission time k over startup overhead t_0. P is the number of processors. (See also Sections 9.4.3 and 9.4.5)

whenever the input size is a multiple of the cache size. However, one should resist the temptation to use a linear scale when x-values over many orders of magnitude are important but the own results look particularly good for large inputs.

Sometimes, transformations of the x-axis other than linear or logarithmic make sense. For example, in queuing systems one is often interested in the delay of requests as the system load approaches the maximum performance of the system. Figure 9.2 gives an example. Assume we have a disk server with 64 disks. Data is placed randomly on these disks using a hash function. Assume that retrieving a block from a disk takes one time unit and that there is a periodic stream of requests — one every $(1 + \epsilon)/64$ time units. Using queuing theory one can show that the delay of a request is approximately proportional to $1/\epsilon$ if only one copy of every block is available. Therefore, it makes sense to use $1/\epsilon$ as the x-value. First, this transformation makes it easy to check whether the system measured also shows this behavior linear in $1/\epsilon$. Second, one gets high resolution for arrival rates near the saturation point of the system. Such high arrival rates are often more interesting than low arrival rates because they correspond to very efficient uses of the system.

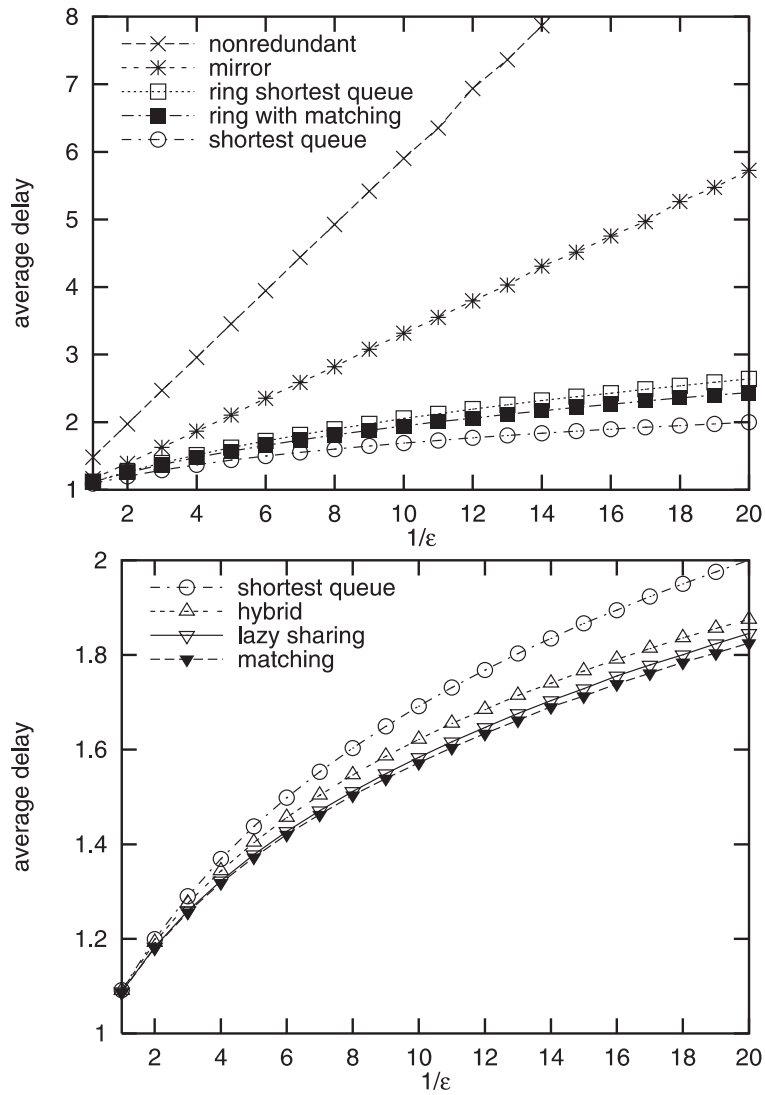


Fig. 9.2. Comparison of eight algorithms for scheduling accesses to parallel disks using the model described in the text (note that "shortest queue" appears in both figures). Only the two algorithms "nonredundant" and "mirror" exhibit a linear behavior of the access delay predicted by queuing theory. The four best algorithms are based on *random duplicate allocation* — every block is available on two randomly chosen disks and a scheduling algorithm [9.13] decides which copy to retrieve. (See also Section 9.4.3)

9.4.2 The *y*-Axis

Given that the x-axis often has a logarithmic scale, we often seem to be forced to use a logarithmic scale also for the y-axis. For example, if the execution time is approximately some power of the problem size, such a double-logarithmic plot will yield a straight line.

However, plots of the execution time can be quite boring. Often, we already know the general shape of the curve. For example, a theoretical analysis may tell us that the execution time is between $T(n) = \Omega(n)$ and $T(n) = \mathcal{O}(n\text{polylog}(n))$. A double-logarithmic plot will show something very close to a diagonal and discerns very little about the polylog term we are really interested in. In such a situation, we transform the y-axis so that a priori information is factored out. In our example above we could better display $T(n)/n$ and then use a linear scale for the y-axis. A disadvantage of such transformations is that they may be difficult to explain. However, often this problem can be solved by finding a good term describing the quantity displayed. For example, "time per element" when one divides by the input size, "competitive ratio" when one divides by a lower bound, or "efficiency" when one displays the ratio between an upper performance bound and the measured performance. Figure 9.3 gives an example for using such a ratio.

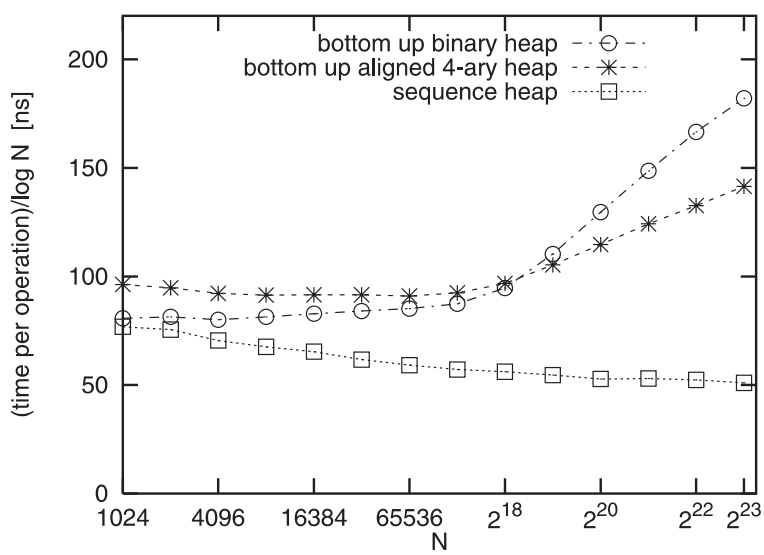


Fig. 9.3. Comparison of three different priority queue algorithms [9.16] on a MIPS R10000 processor. N is the size of the queue. All algorithms use $\Theta(\log N)$ key comparisons per operation. The y-axis shows the total execution time for some particular operation sequence divided by the number of deletion/insertion pairs and $\log N$. Hence the plotted value is proportional to the execution time per key comparison. This scaling was chosen to expose cache effects which are now the main source of variation in the y-value. (See also Sections 9.4.1 and 9.4.3.)

Another consideration is the range of y-values displayed. Assume $y_{\min} > 0$ is the minimal value observed and $y_{\max}$ is the maximal value observed. Then one will usually choose $[y_{\min}, y_{\max}]$ or (better) a somewhat larger interval as the displayed range. In this case, one should be careful however with overinterpreting the resulting picture. A change of the y-value by 1 % will look equal to a change of y-value of 400 %. If one wants to support claims such as "for large x the improvements due to the new algorithm become very large" using a graph, choosing the range $[0, y_{\max}]$ can be a more sound choice. (At least if $y_{\max}/y_{\min}$ is not too close to one. Some of the space "wasted" this way can often be used for placing curve labels.) In Figure 9.2, using $y_{\min} = 1$ is appropriate since no request can get an access delay below one in the model used.

The choice of the the maximum y value displayed can also be nontrivial. In particular, it may be appropriate to clip extreme values if they correspond to measurement points which are clearly useless in practice. For example, in Figure 9.2 it is not very interesting to see the entire curve for the algorithm "nonredundant" since it is clearly outclassed for large $1/\epsilon$ anyway and since we have a good theoretical understanding of this particular curve.

A further degree of freedom is the vertical size of the graph. This parameter can be used to achieve the above goals and the rule of "banking to 45°": The weighted average of the slants of the line segments in the figure should be about 45°.[2] Refer to [9.5] for a detailed discussion. The weight of a segment is the x-interval bridged. There is good empirical and mathematical evidence that graphs using this rule make changes in slope most easily visible.

If banking to 45° does not yield a clear insight regarding the graph size, a good rule of thumb is to make the graph a bit wider than high [9.17]. A traditional choice is to use the golden ratio, i.e., a graph that is 1.62 times wider than high.

9.4.3 Arranging Multiple Curves

An important feature of two-dimensional graphs is that we can place several curves in a single graph as in Figures 9.1, 9.2, and 9.3. In this way we can obtain a high information density without the disadvantages of three-dimensional plots. However, one can easily overdo it resulting in a chaos of undecipherable points and lines. How many curves fit into one pictures depends on the information density. When curves are very smooth, and have few points where they cross each other, as in Figure 9.2, up to seven curves may fit in one figure. If curves are very complicated, even three curves may be too much. Often one will start with a straight-forward graph that turns out to be too ugly for publication. Then one can use a number of techniques to improve it:

[2] This is one of the few things described here which are are not easy to do with gnuplot. But even keeping the principle of banking to 45° in mind is helpful.

- Remove unnecessary curves. For example, Figure 9.2 from [9.13] compares only eight algorithms out of eleven studied in this paper. The remaining three are clearly outclassed or equivalent to other algorithms for the measurement considered.
- If several curves are too close together in an important range of x-values, consider using another y range or scale. If the small differences persist and are important, consider to use a separate graph with a magnification. For example, in Figure 9.2 the four fastest algorithms were put into a separate plot to show the differences between them.
- Check whether several curves can be combined into one curve. For example, assume we want to compare a new improved algorithm with several inferior old algorithms for input sizes on the x-axis. Then it might be sufficient to plot the speedup of the new algorithm over the best of the old algorithms; perhaps labeling the sections of the speedup curve so that the best of the old algorithms can be identified for all x-values. Figure 9.1 gives an example where the speeup of one algorithm over two other algorithms is shown.
- Decrease noise in the data as described in Section 9.4.6.
- Once noise is small, replace error bars with specifications of the accuracy in the caption as in Figure 9.6.
- Connect points belonging to the same curves using straight lines.
- Choose different point styles and line styles for different curves.
- Arrange labels explaining point and line styles in the "same order"[3] as they appear in the graph. Sometimes one can also place the labels directly at the curves. But even then the labels should not obscure the curves. Unfortunately, gnuplot does not have this feature so that we could not use it in this paper.
- Choose the x-range and the density of x-values appropriately.

Sometimes we need so many curves that they cannot fit into one figure. For example, when the cross-product of several parameter ranges defines the set of curves needed. Then we may finally decide to use several figures. In this case, the same y-ranges should usually be chosen so that the results remain comparable. Also one should choose the same point styles and line styles for related curves in different figures, e.g., for curves belonging to the same algorithm as for the "shortest queue" algorithm in Figure 9.2. Note that tools such as gnuplot cannot do that automatically.

The explanations of point and line styles should avoid cryptic abbreviations whenever possible and at the same time avoid overlapping the curves. Both requirements can be reconciled by placing the explanations appropriately. For example, in computer science, curves often go from the lower left corner to the upper right corner. In that case, the best place for the definition of line and point styles is the upper left corner.

[3] For example, one could use the order of the y-values at the larges x-value as in Figure 9.3.

9.4.4 Arranging Instances

If measurements like execution time for a small set of problem instances are to be displayed, a bar chart is an appropriate tool. If other parameters such as the algorithm used, or the time consumed by different parts of the algorithm should be differentiated, the bars can be augmented to encode this. For example, several bars can be stacked in depth using three-dimensional effects or different pieces of a bar can get different shadings.[4]

If there are so many instances that bar charts consume too much space, a *scatter plot* can be useful. The x-axis stands for a parameter like problem size and we plot one point for every problem instance. Figure 9.4 gives a simple example. Point styles and colors can be used to differentiate different types of instances or variations of other parameters such as the algorithm used. Sometimes these points are falsely connected by lines. This should be avoided. It not only looks confusing but also wrongly suggests a relation between the data points that does not exist.

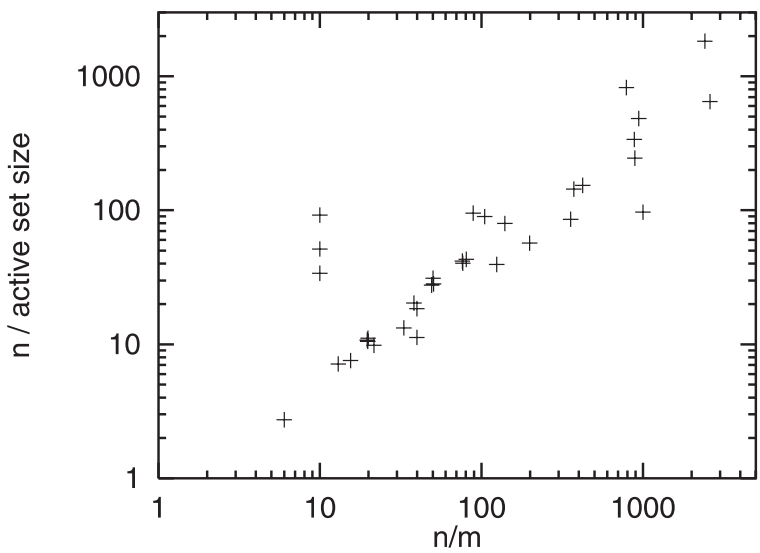


Fig. 9.4. Each point gives the ratio between total problem size and "core" problem size in a fast algorithm for solving set covering problems from air line crew scheduling [9.1]. The larger this ratio, the larger the possible speedup for a new algorithm. The x-axis is the ratio between the number of variables and number of constraints. This scale was chosen to show that there is a correlation between these two ratios that is helpful in understanding when the new algorithm is particularly useful. The deviating points at $n/m = 10$ are artificial problems rather different from typical crew scheduling problems. (See also Section 9.4.1.)

[4] Sophisticated fill styles give us additional opportunities for diversification but Tufte notes that they are often too distracting [9.17].

9.4.5 How to Connect Measurements

Tools such as gnuplot allow us to associate a measured value with a symbol like a cross or a star that clearly specifies that point and encodes some additional information about the measurement. For example, one will usually choose one point symbol for each displayed curve. Additionally, points belonging to the same curve can be connected by a straight line. Such lines should usually not be viewed as a claim that they present a good interpolation of the curve but just as a visual aid to find points that belong together. In this case, it is important that the points are large enough to stand out against the connecting lines. An alternative is to plot measurements points plus curves stemming from an analytic model as in Figure 9.5.

The situation is different if only lines and no points are plotted as in Figure 9.1. In this case, it is often impossible to tell which points have been measured. Hence such a lines-only plot implies the very strong claim that the points where we measured are irrelevant and the plotted curve is an accurate representation of the true behavior for the entire x-range. This only makes sense if very dense measurements have been performed and they indeed form a smooth line. Sometimes one sees smooth lines that are weighted averages over a neighborhood in the x-coordinates. Then one often uses very small points for the actual measurements that form a cloud around this curve.

A related approach is connecting measured points with interpolated curves such as splines which are more smooth than lines. Such curves should only be used if we actually conjecture that the interpolation used is close to the truth.

9.4.6 Measurement Errors

Tools allow us to generalize measured points to ranges which are usually a point plus an error bar specifying positive and negative deviations from the y-value.[5] The main question from the point of view of designing graphs is what kind of deviations should be displayed or how one can avoid the necessity for error bars entirely.

Let us start with the well behaved case that we are simulating a randomized algorithm or work with randomly generated problem instances. In this case, the results from repeated runs are independent identically distributed random variables. In this case, powerful methods from statistics can be invoked. For example, the point itself may be the average of the measured values and the error bar could be the standard deviation or the standard error [9.11]. Figure 9.5 gives an example. Note that the latter less well known quantity is a better estimate for the difference between the average and the actual mean. By monitoring the standard error during the simulation, we can

[5] Uncertainties in both x and y-values can also be specified but this case seems to be rare in Algorithmics.

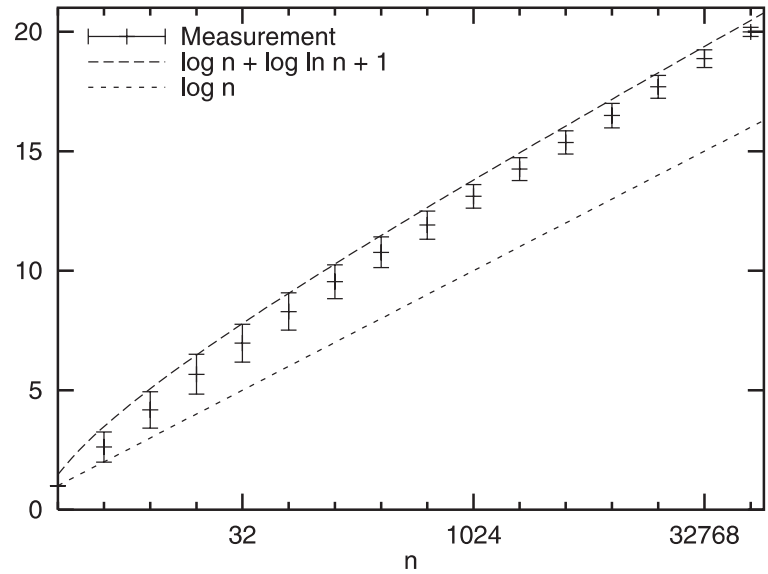


Fig. 9.5. Number of iterations that the dynamic load balancing algorithm *random polling* spends in its warmup phase until all processors are busy. Hypothesized upper bound, lower bound and measured averages with standard deviation [9.12, 9.14]. (See also Sections 9.4.1 and 9.4.5.)

even repeat the measurement sufficiently often so that this error measure is below some prespecified value. In this case, no error bars are needed and it suffices to state the bound on the error in the caption of the graph. Figure 9.6 gives an example.

The situation is more complicated for measurements of actual running times of deterministic algorithms, since this involves errors which are not of a statistical nature. Rather, the errors can stem from hidden variables such as operating system interrupts, which we cannot fully control. In this case, points and error bars based on order statistics might be more robust. For example, the y value could be the median of the measured values and the error bar could define the minimum and the maximum value measured or values exceeded in less than 5 % of the measurements. The caption should explain how many measurements have been performed.

9.5 Grids and Ticks

Tools for drawing graphs give us a lot of control over how axes are decorated with numbers, tick marks and grid lines. The general rule that is often achieved automatically is to use a few round numbers on each axis and perhaps additional tick marks without numbers. The density of these numbers

should not be too high. Not only should they appear well separated but they also should be far from dominating the visual appearance of the graph. When a very large range of values is displayed, we sometimes have to force the system to use exponential notation on a part of the axis before numbers get too long. Figure 9.6 gives an example for the particularly important case of base two scales. Sometimes we may decide that reading off values is so important in a particular graph that grid lines should be added, i.e., horizontal and vertical lines that span the entire range of the graph. Care must be taken that such grid lines to not dilute the visual impression of the data points. Hence, grid lines should be avoided or at least made thin or, even better, light gray. Sometimes grid lines can be avoided by plotting the values corresponding to some particularly important data points also on the axes.

A principle behind many of the above considerations is called Data-Ink Maximization by Tufte [9.17]. In particular, one should reduce non-data ink and redundant data ink from the graph. The ratio of data ink to total ink used should be close to one. This principle also explains more obvious sins like pseudo-3D bar charts, complex fill styles, etc.

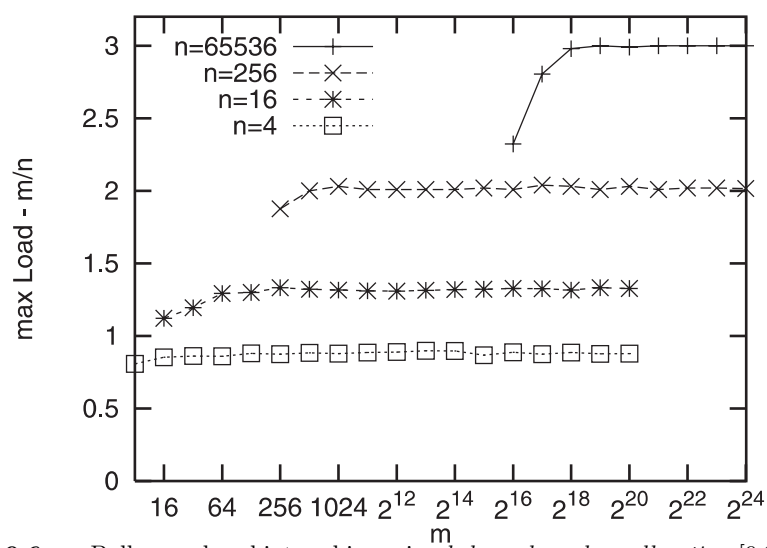


Fig. 9.6. m Balls are placed into n bins using *balanced random allocation* [9.2, 9.3]. The difference between maximal and average load is plotted for different values of m and n. The experiments have been repeated at least sufficiently often to reduce the *standard error* ($\sigma/\sqrt{\text{repetitions}}$ [9.11]) below one percent. In order to minimize artifacts of the random number generator, we have used a generator with good reputation and very long period ($2^{19937} - 1$) [9.7]. In addition, some experiments were repeated with the Unix generator `srand48` leading to almost identical results. (See also Section 9.4.3.)

9.6 Three-Dimensional Figures

On the first glance, three-dimensional figures are attractive because they look sophisticated and promise to present large amounts of data in a compact way. However there are many drawbacks.

- It is almost impossible to read absolute values from the two-dimensional projection of a function.
- In complicated functions interesting parts may be hidden from view.
- If several functions are to be compared, one is tempted to use a corresponding number of three-dimensional figures. But in this case, it is more difficult to interpret differences than in two-dimensional figures with cross-sections of all the functions.

It seems that three-dimensional figures only make sense if we want to present the general shape of a single function. Perhaps three-dimensional figures become more interesting using advanced interactive media where the user is free to choose viewpoints, read off precise values, view subsets of curves, etc.

9.7 The Caption

Graphs are usually put into "floating figures" which are placed by the text formatter so that page breaks are taken into account. These figures have a caption text at their bottom which makes the figure sufficiently self contained. The captions explains *what* is displayed and *how* the measurements have been obtained. This includes the instances measured, the algorithms and their parameters used, and, if relevant the system configuration (hardware, compiler,...). One should keep in mind that experiments in a scientific paper should be reproducible, i.e., the information available should suffice to repeat a similar experiment with similar results. Since the caption should not become too long it usually contains explicit or implicit references to surrounding text, literature or web resources.

9.8 A Check List

In the following we summarize the rules discussed above. This list has the additional beneficial effect to serve as a check list one can refer to for preparing graphs and for teaching. The Section numbers containing a more detailed discussion are appended in brackets. The order of the rules has been chosen so that in most cases they can be applied in the order given.

- Should the experimental setup from the exploratory phase be redesigned to increase conciseness or accuracy? (9.2)

- What parameters should be varied? What variables should be measured? How are parameters chosen that cannot be varied? (9.2)
- Can tables be converted into curves, bar charts, scatter plots or any other useful graphics? (9.3, 9.4.4)
- Should tables be added in an appendix or on a web page? (9.3)
- Should a 3D-plot be replaced by collections of 2D-curves? (9.6)
- Can we reduce the number of curves to be displayed? (9.4.3)
- How many figures are needed? (9.4.3)
- Scale the x-axis to make y-values independent of some parameters? (9.4.1)
- Should the x-axis have a logarithmic scale? If so, do the x-values used for measuring have the same basis as the tick marks? (9.4.1)
- Should the x-axis be transformed to magnify interesting subranges? (9.4.1)
- Is the range of x-values adequate? (9.4.1)
- Do we have measurements for the right x-values, i.e., nowhere too dense or too sparse? (9.4.1)
- Should the y-axis be transformed to make the interesting part of the data more visible? (9.4.2)
- Should the y-axis have a logarithmic scale? (9.4.2)
- Is it misleading to start the y-range at the smallest measured value? (9.4.2)
- Clip the range of y-values to exclude useless parts of curves? (9.4.2)
- Can we use banking to 45°? (9.4.2)
- Are all curves sufficiently well separated? (9.4.3)
- Can noise be reduced using more accurate measurements? (9.4.3)
- Are error bars needed? If so, what should they indicate? Remember that measurement errors are usually *not* random variables. (9.4.6, 9.4.3)
- Use points to indicate for which x-values actual data is available. (9.4.5)
- Connect points belonging to the same curve. (9.4.3,9.4.5)
- Only use splines for connecting points if interpolation is sensible. (9.4.3,9.4.5)
- Do not connect points belonging to unrelated problem instances. (9.4.5)
- Use different point and line styles for different curves. (9.4.3)
- Use the same styles for corresponding curves in different graphs. (9.4.3)
- Place labels defining point and line styles in the right order and without concealing the curves. (9.4.3)
- Captions should make figures self contained. (9.7)
- Give enough information to make experiments reproducible. (9.7)

References

9.1 P. Alefragis, P. Sanders, T. Takkula, and D. Wedelin. Parallel integer optimization for crew scheduling. *Annals of Operations Research*, 99(1):141–166, 2000.

9.2 Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, February 2000.

9.3 P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *32th Annual ACM Symposium on Theory of Computing (STOC'00)*, pages 745–754, 2000.
9.4 J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*. Duxbury Press, Boston, 1983.
9.5 W. S. Cleveland. *Elements of Graphing Data*. Wadsworth, Monterey, Ca, 2nd edition, 1994.
9.6 D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In M. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002.
9.7 M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.
9.8 C. C. McGeoch, D. Precup, and P. R. Cohen. How to find big-oh in your data set (and how not to). In *Advances in Intelligent Data Analysis*. Springer Lecture Notes in Computer Science 1280, pages 41–52, 1997.
9.9 C. C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.
9.10 B. M. E. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000. To appear.
9.11 W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.
9.12 P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.
9.13 P. Sanders. Asynchronous scheduling of redundant disk arrays. In *12th ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*, pages 89–98, 2000.
9.14 P. Sanders and R. Fleischer. Asymptotic complexity from experiments? A case study for randomized algorithms. In *Proceedings of the 4th Workshop on Algorithm Engineering (WAE'00)*. Springer Lecture Notes in Computer Science 1982, pages 135–146, 2000.
9.15 P. Sanders and J. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. In *Proceedings of the 6th International Euro-Par Conference*. Springer Lecture Notes in Computer Science 1900, pages 918–926, 2000.
9.16 P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
9.17 E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, U.S.A., 1983.

10. Distributed Algorithm Engineering

Paul G. Spirakis and Christos D. Zaroliagis

[1] Computer Technology Institute
P.O. Box 1122, 26110 Patras, Greece

[2] Department of Computer Engineering & Informatics
University of Patras, 26500 Patras, Greece
spirakis@cti.gr
zaro@ceid.upatras.gr

Summary.

When one engineers distributed algorithms, some special characteristics arise that are different from conventional (sequential or parallel) computing paradigms. These characteristics include: the need for either a scalable real network environment or a platform supporting a simulated distributed environment; the need to incorporate asynchrony, where arbitrary asynchrony is hard, if not impossible, to implement; and the generation of "difficult" input instances which is a particular challenge. In this work, we identify some of the methodological issues required to address the above characteristics in distributed algorithm engineering and illustrate certain approaches to tackle them via case studies. Our discussion begins by addressing the need of a simulation environment and how asynchrony is incorporated when experimenting with distributed algorithms. We then proceed by suggesting two methods for generating difficult input instances for distributed experiments, namely a game-theoretic one and another based on simulations of adversarial arguments or lower bound proofs. We give examples of the experimental analysis of a pursuit-evasion protocol and of a shared memory problem in order to demonstrate these ideas. We then address a particularly interesting case of conducting experiments with algorithms for mobile computing and tackle the important issue of motion of processes in this context. We discuss the two-tier principle as well as a concurrent random walks approach on an explicit representation of motions in ad-hoc mobile networks, which allow at least for average-case analysis and measurements and may give worst-case inputs in some cases. Finally, we discuss a useful interplay between theory and practice that arise in modeling attack propagation in networks.

10.1 Introduction

It is a common feeling among scientists, not only in the algorithms community, that a significant fraction of the research done in the algorithms area is eminently practical. However, only a small part of it is actually used. A suggested and also widely accepted remedy of this is that algorithmic research must include experiments and implementation if the field wants to have maximum impact.

In certain new fields much affected by current technology, the need of demonstration of practicality of algorithmic research is more intense. Such

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 197–228, 2002.
© Springer-Verlag Berlin Heidelberg 2002

a field is that of *distributed systems*. These systems are ubiquitous today throughout business, academia, government, and home. Typically, they provide means to share resources and data. More ambitious distributed systems attempt to provide improved performance by attacking subproblems in parallel, and to provide improved availability in case of failures of some components.

The research in *distributed algorithms* tries to identify fundamental problems that are abstractions of those that arise in a variety of distributed systems, state them precisely, and then design and analyze efficient solutions. However, there are some important differences from the sequential case. First, there is not a single, universally accepted model of distributed computation — and there probably never will be — since distributed systems tend to vary much more than sequential computers do. Second, fundamental difficulties are introduced by three factors: *asynchrony, limited local knowledge*, and *failures*. The term asynchrony means that the absolute and even relative times at which events take place cannot always be known precisely. Also, since each computing entity can only be aware of information that it acquires, it has only a local view of the global situation. Computing entities can fail independently, leaving some components operational while others are not.

The explosive growth of distributed systems makes it imperative to understand how to overcome these difficulties. The fact that these difficulties are of a fundamental nature, led the theoreticians of distributed computing towards an effort to abstract and model their "negative" nature. In fact, the field of the theoretical aspects of distributed computing is full of lower bound and impossibility results. It is perhaps the field where the notion of an *adversary* to the solution is so well examined and modeled.

We believe that for all these reasons a systematic theory of *distributed algorithm engineering* should arise. By the term *distributed algorithm engineering* we mean the considerable effort required to convert theoretically efficient and correct distributed algorithms to efficient, robust, and easily used software implementations on a simulated or real distributed environment, usually accompanied by thorough experimentation, fine-tuning and testing. This conversion has to preserve the assumed properties and limitations of the distributed computing model. This in addition implies that the semantics of the implementation operations must agree with those assumed by the theoretical algorithm. Our experience indicates that such a conversion process may lead to improved distributed algorithms through perhaps the experimental discovery of behaviours or properties that were not exploited in the initial theoretical version of the algorithm.

The conducted experiments with implementations of distributed algorithms, henceforth *distributed experiments*, should evaluate algorithms by primarily providing them with "difficult", or "practical" inputs. It is exactly the rich collection of negative results about the "adversaries" of a distributed algorithm that allow the start of a systematic theory of the construction of

"hard" instances, so useful for experiments. In addition, the need for detailed parameterization of the various complexity measures involved in any distributed problem, has led to a good understanding of various costs (such as number of messages, size and number of shared variables, number of faulty components, etc) which provides strong tools for the answer to the important question of "what to measure?" in a distributed experiment. The above idiosyncracies of distributed experiments justify well the creation of a whole new subfield of algorithm engineering. The current paper is a first contribution in this direction.

Our aim in this work is to address several methodological issues in distributed algorithm engineering. We do not attempt to cover every possible issue, but to address those which we find important. As explained above, we emphasize on issues that are not usually encountered when engineering sequential or parallel algorithms. Most of them are illustrated by case studies that are based on our own experience with developing simulators for distributed algorithms and experimenting with implementations of distributed algorithms on these simulators.

We start by discussing the need for a simulation environment which addresses the critical issue of scalability. Real distributed systems and algorithms are asynchronous. Incorporating asynchrony into a simulator is rather hard (if at all possible). To this end, we focus on a causality-affects relation which distributed experiments should obey and discuss advantages and disadvantages of known approaches to achieve it.

We then address the challenging issue of generating difficult (e.g., worst-case) inputs for implementations of distributed algorithms. We argue for two approaches, namely the construction of worst-case event schedules by mimicking impossibility arguments or lower bound proofs, and the use of game-theoretic notions (worst-case Nash equilibria) to construct test-sets which force the implemented algorithm to exhibit a nearly worst-case behaviour. To demonstrate these ideas, we present the experimental analysis of a pursuit-evasion protocol and an example of a lower bound proof for a shared memory problem which leads to worst-case schedules.

We then proceed to a particular interesting case of conducting experiments with algorithms for mobile computing. Two alternative models for mobile computing are discussed, the fixed backbone model and the ad-hoc model. The new element here is the question of how to implement motions. We discuss the two-tier principle that usually guides the fixed backbone model as well as a concurrent random walks approach on an explicit representation of motions in ad-hoc mobile networks which allow at least for average-case analysis and measurements, and may give worst-case inputs in some cases.

Finally, we consider a rather general model for modeling attacks in computer networks and discuss the development of protocols for propagation of attacks under this model. The development of such protocols turns out to be

an interesting case of distributed algorithm engineering as both analytic and experimental methods are used which are tied to each other.

10.2 The Need of a Simulation Environment

In this section, we will argue about the need for a simulation environment which addresses the crucial issue of scalability. We first attempt to formally define the simulator and then give an overview of existing systems.

Distributed applications code runs usually on rather huge networks of possibly heterogeneous local machines. Even if one manages to control such an environment for conducting experiments, still several important questions cannot be answered due to the implied restrictions of available technology. Perhaps the most important one is that of *scalability*: given that a distributed algorithm behaves "well", for example, on a network of ten machines, how will it behave on a much larger network? Thus, this critical issue of scalability of distributed solutions can be experimentally treated only via *simulations*. It is a nice byproduct of distributed algorithmic practice the fact that simulations and simulation environments are themselves extensively studied and formalized. The crucial idea here is that the simulator executions should not alter the nature of executions on the "real" (or envisioned) system. We can capture this via some definitions by adopting the framework in [10.3].

We view a *distributed system* as a collection of a set of *nodes* or *processors*, a communication system $\mathcal{C}$ linking the nodes, and the external *environment* $\mathcal{E}$. Usually the environment $\mathcal{E}$ and the communication system $\mathcal{C}$ are not explicitly modeled but are given as problem specifications, which impose conditions on their behaviour. A *node* or *processor* is a (rather) hardware notion. On each node there are one or more (software) *processes* running. Let us, for the sake of definition of simulations, restrict our attention to the situation where processes are organized into a single *stack of layers* and that there are the same number of layers on each node. Each layer communicates with the layer above it and the layer below it. The bottom layer communicates with $\mathcal{C}$ and the top layer communicates with $\mathcal{E}$.

Each *process* is modeled as an automaton with a (possibly infinite) set of states. Transitions between states are triggered by the occurrence of *events* of the process. *Events* are inputs or outputs that come from (or go to) the layer above or below. A *configuration* of a distributed system specifies a state for every process on every node. An *initial configuration* contains all initial states. An *execution* of a distributed program is a sequence $C_0\Phi_1C_1\Phi_2C_2\cdots$ of alternating configurations C_i and events Φ_i beginning with a configuration and, if finite, ending with a configuration. An execution must satisfy four conditions:

1. C_0 is an initial configuration.
2. For every $i \geq 1$, event Φ_i is enabled in configuration C_{i-1} and configuration C_i is the result of Φ_i acting on C_{i-1}. In more detail, every state is the same in C_i and C_{i-1} except for the (at most two) processes for which Φ_i is an event. The states of these processes change according to the transition functions of those processes.
3. For every $i \geq 1$, if Φ_i is not a node input, then $i > 1$ and Φ_i is on the same node as event Φ_{i-1}.
4. A node input does not happen until all other events have acted and no more are enabled.

The last two conditions are stated just to guarantee atomicity with respect to events on different nodes. A node is triggered into action by the occurrence of an input either from the external environment, or from the communication system. The trigger causes a "chain reaction" of events at the same node, and this occurs atomically, until no more events are enabled, other than node inputs. In fact, there are many ways to state (or even omit) conditions (3) and (4) if the implementation guarantees atomicity of events on different nodes via some other mechanism, for example, via the event generation scheme or via the scheme that assigns durations to steps of processes and delays to messages according to a global simulator (virtual) clock.

The *schedule* of an execution is the sequence of events in the execution. Given execution a, let us denote by $top(a)$ (resp. $bot(a)$) the restriction of the schedule for a to the events on the interface of the top (resp. bottom) layer. An execution a is then said to be *correct for communication system* $\mathcal{C}$ if $bot(a)$ is an element of the allowable sequences of inputs/outputs of $\mathcal{C}$. An execution is *fair* if every event (other than a node input) that is continuously enabled, eventually occurs. This ensures that executions do not halt prematurely, while there is still a step to be taken. An execution a is *user-compliant for problem specification* P if, informally speaking, the environment satisfies the input constraints of P (if any). The details of the input constraints will naturally vary depending on the particular problem. An execution is called $(P, \mathcal{C})$-*admissible* if it is fair, user-compliant for the problem specification P, and correct for the communication system $\mathcal{C}$.

We are now ready to define what a simulator should do. Let us denote by communication system C_1 whatever is available for our experiments. We wish this to simulate some (larger or envisioned) communication system C_2. For such a simulation we then demand the existence of a collection of processes called *Sim* (the simulation program) which must satisfy three laws:

1. The top interface of *Sim* is the interface of C_2.
2. The bottom interface of *Sim* is the interface of C_1.
3. For every (C_2, C_1)-admissible execution a of *Sim*, there exists a sequence σ of events in the set of sequences of events of C_2 such that $\sigma = top(a)$.

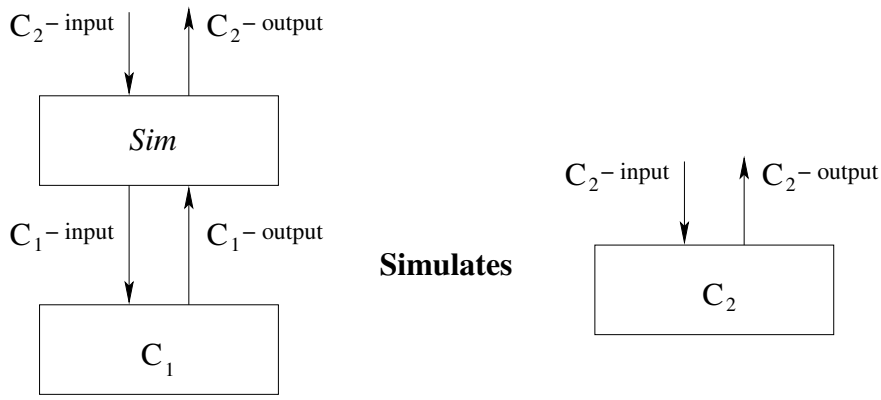


Fig. 10.1. The simulation of a communication system C_2

Informally, we run our simulation on top of the (available for experimentation) communication system C_1 and this produces the same appearance to the external environment as does the (envisioned) communication system C_2. This is illustrated in Figure 10.1.

Let us define the *richness* of (Sim, C_1) as the set of all possible C_2 communication systems that can be simulated by it. It is then apparent that our experimental environment (Sim, C_1) for distributed experiments should be rich enough to include C_2 communication systems which are, for example, scalable C_1 communication systems or extensions or systems with stronger technological capabilities. There are several such environments, some of which we discuss next.

10.2.1 An Overview of Existing Simulation Environments

Simulation environments that provide all the necessary primitives to allow simulation of any distributed algorithm are not so many. To the best of our knowledge, there are three such environments: DSP [10.11], IOA [10.20], and DAP [10.10]. On the other hand, there are several environments for the simulation of network algorithms (i.e., distributed protocols with low-level functionality), or environments that focus on the simulation of a specific area of research in distributed computing. Important examples of such environments include ns [10.32], YACSIM [10.22], and SimUTC [10.42]. Another crucial characteristic is that some simulation environments request that a user develops a protocol using a specific description language provided by the environment, or a scripting language (this is the case for ns, IOA, DSP). This should be contrasted to simulation environments (e.g., YACSIM, SimUTC, DAP) that allow users to develop programs in a standard programming language (`C` or `C++`), which could be advantageous in the sense that the same program can be directly run on a real distributed environment. In the rest of

this section, we shall give a brief overview of the above mentioned simulation environments.

Ns (network simulator) [10.32] is a discrete event simulator aiming at simulating network protocols for low-level functionality, that is, simulation of TCP-like protocols, as well as of routing and multicast protocols over wired and wireless (local and satellite) networks. Ns requires that the protocols and the simulation setup is written in the OTcl scripting language (Tcl with object-oriented extensions by MIT). A user may develop protocols in a standard programming language (`C++`), but needs to bind them to OTcl in order to be simulated by ns. An animation tool (nam) for animating the simulation accompanies ns.

The IOA project [10.20] provides a formal language for describing processes that are modeled using I/O automata and a toolset (currently under development) which will provide support for the development, analysis, and simulation of IOA programs. The toolset is developed in `Java`. To model a distributed algorithm or system, a user has to program in the IOA language. (A similar effort was done earlier with the Esterel language [10.12].)

The Distributed Systems Platform (DSP) [10.11] is a software platform that has been designed for the implementation, simulation, and testing of distributed algorithms. It offers a set of subtools which allow the researcher and the algorithm designer to work under a familiar graphical and algorithmic environment. DSP provides a set of simple, algorithmic languages (DSPL) which can describe the topology and the behaviour of distributed systems and it can support the testing process (on-line simulation management, selective tracing, and presentation of results) during the execution of specific and complex simulation scenarios. The DSP tool has been implemented in `C`.

YACSIM [10.22] is a discrete event simulator implemented in `C` that provides a basic set of `C` subroutines (which model processes, events, queues) that the user can link with his/her program in order to produce an executable that performs the simulation (i.e., YACSIM does not provide a separate simulator, but the simulator is contained in the executable produced by the user). YACSIM is normally used as a base to produce more sophisticated simulators. For example, it was used to produce NETSIM [10.21], a general purpose interconnection network simulator for parallel architectures.

SimUTC [10.42] is a simulation toolkit intended to develop simulations of algorithms for the specific problem of clock synchronization. It is round-based and built on `C++SIM` [10.29], a toolkit written in `C++` that implements facilities provided in `SIMULA` [10.9]. It uses threads and is a process-oriented, continuous-time discrete-event simulator. It is interesting that the `C++SIM` developers observe that `C++` compilers produce much more efficient code than `SIMULA`, thus resulting in faster simulations.

The Distributed Algorithms Platform (DAP) [10.10] is a software platform, currently under development, aiming to support the implementation, simulation, and testing of distributed algorithms. It is implemented in `C++`

using LEDA [10.33]. To transfer the full power of LEDA to distributed experiments and implementation of distributed algorithms, DAP will be provided as a LEDA Extension Package. The major goal of DAP is to provide a homogeneous environment for the simulation of distributed algorithms, regardless of whether they are designed for wired or for wireless networks. It allows the algorithm designer to implement a protocol using a standard programming language (`C++`), along with the primitives of the DAP library, and hence waive the need for re-writing an existing application program as well as the need for an interpreter. DAP also provides a graphical user interface that allows the execution of simulations, protocol animation, as well as correctness checking.

10.3 Asynchrony in Distributed Experiments

Notions of causality and time play an important role in the design of distributed algorithms. It is often helpful to know the relative order in which events take place in the system. This knowledge can be achieved (and experimentally dealt with), even in *totally asynchronous systems* that have no way of measuring the passage of real time, by observing the causality relations between events.

In many systems, processors have access to real-time measuring devices, for example, to hardware clocks, or by tuning in to a satellite clock, or by reading the time across a communication network. In such cases the experiments become easier, since we only have to provide the logical equivalent of a "global time".

Since executions of a distributed system are sequences of events, they induce a *total order* on all possible events. However, this way of describing executions is experimentally painful since it requires the (frequently intolerable) overhead of submitting our experiment to all possible sequences of events! This is wasteful since it is possible that two computation events by different nodes, which may not influence each other, to be nonetheless arbitrarily ordered by the execution. What is important for our experiment to capture is the *structure of causality* between events.

Consider two events by different processors (nodes) – possibly simulated in our experiment. The only way for one processor to influence another processor is by *sending information* (a *message*) to the other processor. But also note that events can causally influence each other indirectly through other events. Hence, it seems that a successful distributed experiment should capture this essential causality relationship for *every* execution. Consequently, a *causality-affects* relation for execution a is defined as follows (see [10.3, Ch. 8] and also [10.28]).

Given two events Φ_1 and Φ_2 in a, we say that Φ_1 *causally-affects* Φ_2 *in* a, denoted by $\Phi_1 \stackrel{a}{\Longrightarrow} \Phi_2$, if one of the following holds:

(i) Φ_1, Φ_2 are events in the same (sequential) process p_i and Φ_2 follows (occurs after) Φ_1 in a's part of p_i.
(ii) Φ_1 is the "send" event of passing information I from process p_i to another process p_j and Φ_2 is the "receive" event of information I by p_j.
(iii) There exists an event Φ such that $\Phi_1 \stackrel{a}{\Longrightarrow} \Phi$ and $\Phi \stackrel{a}{\Longrightarrow} \Phi_2$.

We conclude that distributed experiments should *respect* the causality-affects relationship.

Even if processors cannot "observe" the above relationship, the experiment has to observe it. Till now, there are two approaches (both unsatisfactory for reasons that we will explain) for dealing experimentally with this. The first approach is to supply the experiment with an *event generator* which produces (sequentially in time) events for various processes that respect causality. The second approach is to assign "time durations" (or delays) to each local process step and to each send-receive event in the experiment. These delays can be just integers and may not necessarily relate to the local simulation hardware clock, but they must respect the specifications of the (possibly hypothetical) system on which the experiment runs via the simulator.

The trouble with both approaches is that usually the specifications of the hypothetical system, on which the distributed algorithm runs, allow for many (sometimes a vast number of) total orders of causally related events. An example is the specification of a totally asynchronous system, where no relation is given in advance between durations of local processor steps or message delays.

The usual experimental answer here is the use of random number generators in order to assign local or message event durations. However, even this approach suffers in two directions. First, each random number selection must specify an interval of possible numbers. This, theoretically, restricts the number of possible event orderings. Second, processes might try to draw conclusions about asynchrony by monitoring locally their event durations. Such a (statistical) monitoring should not draw conclusions about biases or about the mechanism of (pseudo) randomness. Cryptographically secure generators might be an answer to this problem [10.5].

In any case, the experimental implementation of the degree of asynchrony, specified by the hypothetical system specifications, should at least produce those admissible sequences of events that can demonstrate the worst-case behaviour of the implemented algorithm when all other inputs are fixed. This issue is further investigated in the next section.

10.4 Difficult Input Instances for Distributed Experiments

In this section we address the problem of how to generate "hard" input instances for experiments on distributed algorithms or protocols. We focus on two approaches:

(a) An *adversarial-based approach*, taking as point of departure the rich literature about adversarial arguments in distributed computing. These arguments often lead to proofs of impossibility results for the computation of certain tasks or to lower bounds on the (worst-case) performance of any distributed algorithm for a certain problem. We note that quite frequently such arguments are indeed *adversarial constructions* in the sense that they propose particular execution orders of certain events (among the admissible schedules) and/or particular fault patterns that can be produced effectively in an experiment and that have the property of driving any distributed algorithm to its limits as far as worst-case performance is concerned. In fact, many impossibility scenarios can be modified suitably to create hard inputs for the experiments.
(b) A *game-theoretic approach*; namely, to view the distributed protocol as a game in cases where processes or agents may act selfishly or compete to each other for resources. The goal is to select (if computationally possible), among the possibly many Nash equilibria for such games (which are a well-accepted characterization of "rational" behaviour in competition situations), those equilibria that are as worst as possible according to a global system, or *social cost*, criterion. Then, the computed worst-case equilibria strategies are used as the "hard" input instances in the experiments.

In the rest of this section we shall elaborate on these two approaches.

10.4.1 The Adversarial-Based Approach

Any distributed algorithm has to overcome a variety of *adversarial* system behaviours. For example, processes may fail (and perhaps later recover), their states can become corrupted, or even they can behave maliciously. Communication channels can fail, lose or delay messages, or deliver them out of order. Also, shared (memory) objects may fail to respond.

Such behaviours are captured by the notion of an *adversary* to the algorithm. The adversary can select the failure patterns and/or the schedules of events among the admissible schedules. The adversary may control a subset of system's processes. Such processes might relay false information (even deliberately) and can be allowed to conspire.

The precise characterization of the power of the adversary is crucial, because its consequences are either *impossibility results* (that is, no distributed

algorithm can achieve certain goals under such adversaries), or *lower bounds* in worst or average case performance which cannot be improved by any distributed algorithm.

Among the earliest impossibility proofs in distributed computing is the impossibility result for the achievement of distributed consensus in an asynchronous system of processes even with only one faulty process, that is, the well-known result by Fischer et al. [10.13]. (Perhaps the oldest impossibility result for agreement of processes on a value was given by Pease et al. [10.37].)

In the paper by Fischer et al. [10.13], the authors were the first to use a *valency argument* to show that consensus achievement is impossible in a totally asynchronous message-passing system which is allowed to tolerate just one process fault. This fault can be the simplest one, i.e., the faulty process fails by halting permanently at some point (fail-stop model). Valency arguments have become the most widely-used techniques for impossibility proofs in distributed computing. We now give an outline of the valency argument.

Recall that a *configuration* is basically a "snapshot" of a distributed system during the execution of an algorithm. It consists of the state of every process and of the surrounding environment (e.g., messages in transit). A configuration of any consensus algorithm is called *univalent* if every possible execution continuing from that configuration gives the same output value, and *multivalent* otherwise. In the case where the possible output values are just two, then the configuration is called *bivalent*. For example, in the *binary consensus* problem all input values to the processes come from $\{0, 1\}$. To achieve consensus, there are two correctness properties that must be satisfied.

1. Agreement: the output values of all processes are identical.
2. Validity: the output value of each process is the input value of some process.

Now, notice that in the case of input values from $\{0, 1\}$, any consensus protocol must have a bivalent initial configuration. Let e be any event applicable to a bivalent configuration C, let D be the set of configurations reachable from C without applying e, and let $D^* = \{C'' :$ configuration C'' follows from configuration C' by applying e and $C' \in D\}$.

Then, it is proved in [10.13] that D^* contains a bivalent configuration (the proof of this, rather intuitive, statement is very technical, but beautiful). Consequently, any deciding (on a value) execution must go from a bivalent initial configuration to a univalent one, which in turn implies that there should be some single step that goes from a bivalent to a univalent configuration. In [10.13] a particular way is proposed for an execution that avoids such steps and thus leading to an execution that never decides. The execution is constructed in *stages*, starting from an initial configuration. Each stage has one or more steps of some processes. A queue of processes is maintained (initially in an arbitrary order). For each process, a queue of incoming messages is also

maintained. The stage ends with the first process in the queue of processes executing a step, in which, if its message queue was not empty at the start of the stage, then its earliest message is received. This process is then moved to the back of the queue of processes.

Note that this execution can be easily implemented in an experiment, and that in an infinite sequence of stages, every process takes an infinite number of steps and receives every message sent to it. In [10.13] it is then shown that this execution avoids a decision ever being reached, because it always produces bivalent configurations.

Although valency arguments are the most well-known techniques to show impossibility results, other arguments (for example, based on algebraic topology [10.18]) have also been used.

We note that in the valency arguments it is crucial for the adversarial scheduler to select when to schedule a particular process (in order to destroy consensus). Most lower bound arguments also make use of such adversarial schedules of events. Based on this important remark, we propose that such adversarial schedules should be tried (if possible) in a distributed experiment. Then, the experiment will reveal the worst-case behaviour of the proposed protocol under test. Of course, adversarial schedules are not always easy or possible to construct, but the impossibility or lower bound proofs in most cases give strong hints. We illustrate the method through an example.

10.4.1.1 An Example of an Adversarial Schedule. In this section, we shall present an example of an adversarial schedule which is easy to implement.

Suppose that we want to experiment with an algorithm A that solves the following problem, called the *write-all problem* [10.23, 10.25]: P processes are given, all having access to a shared memory (that is, to an array $M[1..\infty]$). Let the first N shared memory locations be called the *write-all array*. All processes are assumed to work in complete synchrony, that is, in each global time unit each process takes a step. The adversary can cause arbitrary process failures and restarts. The problem is to provide a distributed algorithm which, at termination, has managed to *touch* (mark) each position of the write-all array by some process (initially all memory is untouched). The performance measure here is the total number of steps of all processes until this is done. This is called the *work* of the algorithm.

To test such an algorithm A, we suggest the following adversarial failure/restart schedule as proposed in [10.6]. Consider each global step. Let $U > 1$ be the number of untouched array elements (i.e., the elements that no process succeeds in writing to them). For as long as $U > P$ the adversary induces no failures. The work needed to touch $N - P$ array elements when there are no failures is at least $N - P$. As soon as a process is about to touch the element $N - P + 1$, making $U \leq P$, the adversary fails it and then restarts all P processes. For the upcoming cycle, the adversary examines the algorithm's implementation to determine how the processes are assigned to touch ar-

ray elements. The adversary then lists the first $\lfloor U/2 \rfloor$ untouched elements with the least number of processes assigned to them. The total number of processes assigned to these elements cannot exceed $\lceil P/2 \rceil$. Subsequently, the adversary fails these processes and allows all others to proceed. Therefore, at least $\lfloor P/2 \rfloor$ will complete their step having touched no more than half of the remaining untouched array locations. This strategy of failures/restarts can be continued for at least $\log P$ global steps. Then, the work that A performs is at least $N - P + \lfloor P/2 \rfloor \log P = N + \Omega(P \log P)$.

Note that the above schedule of failures/restarts can be easily constructed in the experiment, given any algorithm A and its implementation. Note also that the lower bound to the required work does not count how much work is needed for the processes to read and locally process (without touching the write-all array) the entire shared memory. Thus, such a schedule will cause the implementation of any algorithm A to perform work bounded from below by $N - P + \lfloor P/2 \rfloor \log P$. Most algorithms A will actually do more work, since usually processes can read only a constant number of shared memory locations at each step. Let $L = N - P + \lfloor P/2 \rfloor \log P$ and let $W(A)$ denote the actual work performed by A's implementation. We can then use $W(A) - L$ as a measure of how work-efficient A is for such work-demanding schedules.

Logarithmic lower bounds on the time of any synchronous execution for deterministic write-all were first derived in [10.24].

10.4.2 The Game-Theoretic Approach

Distributed systems often invoke a set of independent *selfish* and *antagonistic agent* processes trying to share a common resource. This situation evokes game theory and its main concept of rational behaviour, the *Nash equilibrium*: in an environment in which each agent is aware of the situation facing all other agents, a Nash equilibrium is a combination of choices (deterministic or randomized), one for each agent, from which no agent has an incentive to unilaterally move away. The ratio between the worst possible Nash equilibrium and the global optimum, called *coordination ratio*, was first defined in [10.27]. Some upper bounds for this ratio and the structure of worst-case Nash equilibria for a very simple routing problem were given in [10.31].

An alternative way to derive "difficult" behaviours (schedules of events) for distributed experiments is to use such game-theoretic ideas. We can sometimes consider the competition between a distributed algorithm and an adversary as a game of possibly many rounds of moves of the opponents. Worst-case Nash equilibria (with respect to some optimization criteria) may then be examined and we suggest them as interesting behaviours to be tested experimentally.

We motivate the above approach by a very simple problem of pursuit-evasion. Several agents moving along neighbor vertices of a graph (network) G are looking for a fugitive. The fugitive is eliminated when it coincides with an agent at a vertex. The agents cannot "see" further away from their current

location. The way the graph (network) operates is a sequence of *rounds*, each of the form $R = (T, F, S)$, where T is an *agent's target phase*, followed by a *fugitive motion phase* F, followed by an *agent's motion phase* S.

Any protocol (strategy) for the agents implements T and S as follows. Let $v(a)$ be the current position, i.e., vertex in G, of agent a, and let $N(v(a))$ denote the set of neighbors of $v(a)$ in G. Let also $t(a)$ be a variable which can be written by the agent and can be read by the fugitive under some conditions that we explain next. We call $t(a)$ the *target variable* of agent a. T is implemented by setting $t(a)$ to be a vertex in $\{v(a)\} \cup N(v(a))$. S is implemented by setting the next position $v'(a)$ of the agent to be the value of $t(a)$.

Any strategy for the fugitive, f, implements F as follows. Let $v(f)$ be the position (vertex in G) of f just before round R, and let $N(v(f))$ denote the set of neighbors of $v(f)$ in G. Each edge (x, y) in G (in direction from x to y) is equipped with a queue, called the *input channel of vertex* y, where an entity located at x can put information. This information can subsequently be read by y. The fugitive can read (and store in its local memory) the value of one input channel $c(u)$ for each $u \in N(v(f))$. The value of each $c(u)$ is defined as follows: if there is an agent a in R with $v(a) = u$ and $t(a) = v(f)$, then $c(u) = v(f)$; otherwise, $c(u)$ is undefined. Thus, the fugitive is allowed to be "warned" about the next position of any agent only when that position is the next position of the fugitive f (we say that the fugitive has a limited sense of approaching agents). Let $\mathcal{C}_i = \{c(u) : \forall u \in N(v(f))$ where $v(f)$ is the position of f in round $i\}$ be the set of all "warnings" that f got in round i. Then, the ordered tuple $H_R(f) = \langle \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_R \rangle$ of channel values of all rounds till R denotes the *history* of all "warnings" that f got up to round R. Consequently, the fugitive's strategy F decides on a next position for f (which must be a neighbor of $v(f)$ in G) based on $H_R(f)$.

Simple randomized protocols for catching the fugitive were presented in [10.39, 10.40]. In their general structure, these protocols suggest that agents are partitioned into two sets: the *traps*, which stay immobile (hidden therefore) at some random vertices of the graph, and the *searchers*, i.e., agents continuously performing independent random walks. Note that the above stated model for the strategies of the agents and the fugitive does not allow the fugitive to "sense" neighboring traps, since their intention variable does not change and thus no information comes via the related channel. Actually, f cannot distinguish between traps and vertices which are neighbors and agent-free.

In this game, good strategies for the fugitive should allow it to stay alive and not be eliminated for as long as possible. Fugitive motion around some chosen cycle in the graph is (together with the way agents act) a Nash equilibrium. The fugitive, while not caught, has no benefit in not following the cycle. The game ends almost surely only if the traps can re-randomize their

locations from time to time [10.40]. Since long cycles of fugitive's movement have higher probability to encounter a trap, we conclude that short cycles of such a movement are worst-case equilibria in the sense that they extend the game's duration. Note that strategies which force the fugitive to stay at some vertex forever (after some initial motion) are not good, since random walks will hit any of those positions in short expected time.

A preliminary set of experiments with such strategies (that is, short cycles in the graph for motions of the fugitive) was performed in [10.15], while the game was theoretically analyzed in [10.39, 10.40]. The experiments indeed demonstrated the longest durations of the game for such fugitive strategies. These experiments were performed on the DSP tool [10.11] and are as follows. First note that if the fugitive is memoryless (or has only a fixed-size memory), then its best strategy is to randomly-avoid approaching agents, that is, when it senses an approaching agent, it should choose randomly any way to go except for the edge via which the agent is approaching. Hence, the experiments become much more interesting when the fugitive can store and remember non-fixed parts of the graph. The fugitive initially wanders in the graph until it discovers a small-sized cycle. Then it stops wandering and starts moving along the cycle.

Various different graph topologies, including regular, irregular and random graphs, were considered in [10.15], each consisting of 100 vertices and the same number of expected edges (about 1000). The protocol was initialized with 5 traps and 2 searching agents. The traps were re-randomizing their positions periodically. The performance was measured in extermination time, that is, after how many simulation rounds (excluding the rounds required for the traps to take their positions) the fugitive falls into a trap. The reported experiments indicated that the cycling fugitive lasts much longer (about 40 times) than the randomly-avoiding fugitive. Experiments were also conducted with various numbers of traps which indicated that when the number of traps is doubled, the extermination time drops almost linearly. A last set of experiments was performed with varying time periods, called *epochs*, for re-randomization of the location of the traps. The experiments showed that the extermination time grows almost linearly with the increase in the epoch size. In all cases the experiments considerably helped the fine-tuning of the parameters of the experimentally best strategy of the agents (number of traps, duration of epochs) against the worst-case equilibrium strategy of the fugitive.

We note that the approach presented above has already led other researchers to design fugitive strategies and agent strategies in similar ways, by especially following the game-theoretic paradigm in slightly different games where the fugitive is completely blind [10.2].

10.5 Mobile Computing

The major technological advances in mobile networks have recently motivated the introduction of a completely new computing environment called *mobile* (or *nomadic*) *computing* [10.19, 10.38].

Mobile computing is a special type of distributed computing that is characterized by four kinds of constraints which make the design of mobile information systems a highly complex task:

(i) Mobile elements have poor resources compared to static elements.
(ii) Mobile elements rely on a finite energy source (battery).
(iii) Mobility is inherently hazardous regarding damage or physical security loss.
(iv) Mobile connectivity is highly variable in performance and reliability.

Hence, information access as well as fundamental distributed computing problems (e.g., leader election) have to be re-considered in the new setting. As a consequence, new approaches are usually required for the effective solution of these problems in order to develop a dependable and efficient mobile information system.

Until now, two basic models have been proposed for mobile computing: the *fixed backbone* model and the *ad-hoc* model. The *fixed backbone* model assumes that a fixed infrastructure of support stations with centralized network management is provided in order to ensure efficient communication. Communication is done through the support stations which serve a certain geographical area in which mobile hosts are moving. The fixed backbone model is motivated by the current status of pragmatic mobile networks.

The *ad-hoc* model assumes that mobile hosts can form temporary networks, called *ad-hoc networks*, without the aid of any fixed infrastructure or centralized administration. Communication between two hosts can be achieved through other mobile hosts which participate in the ad-hoc network and are willing to transfer packets for them. The ad-hoc model is motivated by the need for rapid deployment of mobile hosts in an unknown terrain (e.g., emergency services in a disaster area), where there is no underlying fixed network infrastructure either because it is impossible or very expensive to create it, or because it has become unavailable.

The above imply that the design, implementation, testing and verification of distributed protocols for mobile computing requires extension of software (simulation) platforms that are designed to support development of distributed algorithms on (classical) fixed networks. A simulation environment for mobile computing should be able to capture the notions of a *mobile process* (can be viewed as a virus or an agent), the *motion* of a process or host (which allows it to either migrate from cell to cell or to follow any course in a given space), the *energy* of a process or host, and the *channels* (which except for their bandwidth and latency could be further characterized by their

frequency spectrum as well as by communication interference). All these are crucial in the design of a simulation platform for mobile computing (for example, they are taken into account in the Distributed Algorithms Platform [10.10]).

Perhaps the most important of the above notions is that of *motion* of processes or hosts. Two ways of capturing motions have appeared in the literature: the explicit and the implicit representation. The *explicit* representation provides a definition of the space of possible motions and also a definition of a trajectory of each host in that space. The *implicit* representation provides a dynamic graph of possible direct communications among hosts, and the way this graph changes with time. The change of communication edges is assumed to occur due to the motion of hosts.

One of the most interesting cases is to deal with systems in which fast motions of processes or hosts are allowed. In such cases, the implicit representation has major inherent analysis problems and limitations, since in such dynamic graph models protocols usually try to maintain network structures (e.g., connectivity, multiple paths, etc), but the time to allow information to propagate for the modification of these structures is not always comparable with the speed of change of the network.

Consequently, in the rest of this section, we will focus on the explicit representation of motions and will address certain methodological issues that arise in distributed algorithm engineering regarding implementation and experimentation of algorithms for mobile computing. We shall address these issues through two case studies, one for the fixed backbone model and another for the ad-hoc model. The former concerns the problem of counting the number of mobile hosts in a mobile network, while the latter concerns the fundamental problem of establishing point-to-point communication between two mobile hosts. Before diving into the case studies, we shall discuss the two models in more detail.

10.5.1 Models of Mobile Computing

The *fixed backbone* model assumes two distinct sets of entities in a mobile network: a large number of *mobile hosts*, and a relatively small number of more powerful, fixed hosts. The fixed hosts and the communication paths between them constitute the *static* or *fixed* (part of the) network. The geographical area that is served by the fixed network is divided into smaller regions called *cells*. Each cell is served by a fixed host, also referred to as the *mobile service station* (MSS) of the cell. An MSS communicates with the mobile hosts within its cell via a wireless medium of low bandwidth. Host mobility is represented in this model as migration of mobile hosts between cells; each mobile host belongs to only one cell at any time instance.

We now discuss how mobile hosts exchange messages and which is the incurred cost in the fixed backbone model. There are two types of messages: point-to-point messages between any two MSSs, and messages between a

mobile host and its local MSS. Let the cost of a former message be C_f and the cost of a latter message be C_w. Assume that a mobile host h_1 wants to send a message to another mobile host h_2. The host h_1 sends first the message to its local MSS, which forwards this message to the local MSS currently serving h_2. Since, however, the location of a mobile host is neither fixed nor universally known to the network, the local MSS of h_1 needs first to determine the MSS which currently serves h_2 and then transmit the message. This incurs an extra *search cost* C_s for each message transmission. A reasonable assumption to consider [10.4] is that $C_s = aC_f$, where a is a constant depending on the location management strategy used. Suppose that m mobile hosts are moving throughout a fixed base station mobile network $G = (V, E)$ consisting of $n = |V|$ nodes (corresponding to the MSSs) and $|E|$ edges (representing the point-to-point direct communications between the MSSs). Note that $|E| = O(n^2)$ in the worst-case and that usually $m \gg n$. Let D be the diameter of G. Then, $C_s = O(D)$ and a message between two mobile hosts incurs a cost of $2C_w + C_s$.

The *ad-hoc* model assumes that mobile hosts can form temporary networks, called *ad-hoc networks*. An *ad-hoc mobile network* [10.19] is a collection of mobile hosts with wireless network interfaces forming a temporary network without the aid of any established infrastructure or centralized administration. In an ad-hoc network two hosts that want to communicate may not be within wireless transmission range of each other, but could communicate if other hosts between them are also participating in the ad-hoc network and are willing to forward packets for them.

10.5.2 Basic Protocols in the Fixed Backbone Model

A fundamental problem in any network is to count the number of available processes or nodes. In the case of mobile networks, this *counting problem* retains its importance: the knowledge of how many mobile users are currently connected is generally valuable and can be used both by the control and the application level of the network.

Two algorithms for the counting problem in the fixed backbone model were implemented and experimentally compared in [10.16]. The first algorithm in that paper is a simple modification of an existing algorithm for the counting problem in a fixed network [10.41, Ch. 6]. The second is a more efficient algorithm presented in [10.17].

In the rest of this section, we shall discuss the implementation and experimental evaluation of these algorithms on the Distributed Systems Platform [10.11]. In particular, we discuss the main issue of modeling the speed and the type of movement of the mobile hosts.

For the counting problem, it is assumed that one of the mobile hosts (the *initiator*) wants to find the number of the mobile hosts (m) in the network. It is further assumed that: the communication between the MSSs is based on the asynchronous timing model; an operational mobile host responds immediately

to messages broadcasted by its local MSS; each mobile host has its own distinct identity; the set of mobile hosts does not change during the execution of a counting algorithm.

The first algorithm considered in [10.16], called the *virtual topology algorithm* (VTA), is based on the distributed execution by the mobile hosts of a counting algorithm for fixed networks. The VTA algorithm is based on the assumption that the mobile hosts are willing to control by themselves the execution of the algorithm by avoiding the participation of the MSSs (this may be necessary if some protocol requires computational power that increases the overhead of MSSs). The counting algorithm for fixed networks used is the *Echo protocol* given in [10.41, p.190]. It is based on the centralized wave paradigm [10.41, Ch. 6]. There is one initiator process and all other are non-initiators. The initiator floods token messages to all processes and eventually receives confirmation from all processes. The initiator sends messages to all its neighbors. Upon receipt of the first message, a non-initiator forwards messages to all of its neighbors, except the one from which the message was received and which marks as its parent and the corresponding link as its parent link. It is easy to see that parent links define a spanning tree of the fixed network. When a non-initiator has received messages from all its neighbors, it sends an "echo" message to its parent. When the initiator has received a message (either an echo or a flooding message) from all its neighbors, it terminates. The application of the echo protocol in a mobile network implies that some kind of virtual topology is defined on the mobile hosts. By assuming that the virtual topology has $O(m)$ edges, the total cost of the VTA algorithm is $O(mC_w + DmC_f)$, where D comes from the search cost in the fixed network.

The VTA algorithm has two drawbacks: (i) the high cost of message transmissions in the fixed network; and (ii) it requires the participation of every mobile host in the virtual topology. The latter is crucial, since no mobile host is allowed to disconnect during execution of VTA and which in turn brings into play another parameter: the total execution time of the algorithm which should not be large since otherwise it would increase the consumption of battery power in the mobile hosts.

The second algorithm implemented in [10.16] is a new counting algorithm, especially designed for the fixed backbone model, and presented in [10.17]. The algorithm is based on a common guiding principle of distributed protocols in the fixed backbone model, called the *two tier principle* [10.1], and consequently named the *two tier algorithm* (TTA). The idea of this principle is that the computation and communication costs of an algorithm should be based, as much as possible, on the fixed portion of the mobile network.

The TTA algorithm is also based on the execution of the Echo protocol. The execution is started by the *initiator mobile host* which broadcasts a "count" message (afterwards, this initiator does not respond to any "count" message). The (non-initiator) mobile hosts receive "count" messages from

their local MSS and respond with "count-me" messages in order to be counted by the MSS. The rest of the TTA algorithm is executed by the fixed part of the network (i.e., by the MSSs). Let the *initiator MSS* be the MSS serving the initiator mobile host. The part of the algorithm executed by the MSSs is as follows.

1. The initiator MSS broadcasts a "count" message in its cell and then spreads along the fixed part of the network the request for counting using the Echo protocol. The protocol starts by sending "count-tok" messages to all adjacent MSSs.
2. An MSS, upon receiving a "count-tok" message, broadcasts a "count" message to its cell and waits to collect answers from mobile hosts in this cell. The MSS also forwards a "count-tok" message to its neighbors to continue the execution of the Echo protocol. When the MSS receives a "count-me" message, it increases a local variable sz which stores the number of counted mobile hosts in its cell. If the MSS receives a "join" message (from a mobile host joining its cell), it broadcasts again a "count" message. After the completion of the Echo protocol, all MSSs have been informed about the execution of a counting algorithm and have broadcasted a "count" message in their cell.
3. The initiator MSS starts a second execution of the Echo protocol by sending a "(size-tok,0)" message to its neighbors aiming at collecting the sz variables from all MSSs to the initiator. An MSS terminates the execution of the second Echo protocol when it has received answers from all its children (in the spanning tree) and has consequently "echoed" its sz variable to its parent. After such termination, an MSS stops to broadcast "count" messages when a new mobile hosts enters its cell. After the completion of the second Echo protocol, the initiator MSS knows the total number of mobile hosts in the network (stored in its local sz variable).
4. The initiator MSS broadcasts a "(size,sz)" message in its cell and then starts a third execution of the Echo protocol by sending a "(inform-tok,sz)" message to its neighbors. This third execution aims at informing all mobile hosts about the size of the network. An MSS, upon receiving a "(inform-tok,sz)" message, broadcasts a "(size,sz)" message in its cell and forwards a "(inform-tok,sz)" message to its neighbors to continue the execution of the Echo protocol. If an MSS receives a "join" message, it broadcasts again the "(size,sz)" message. After the completion of the third Echo protocol, all MSSs have broadcasted the size of the mobile network in their cells. Finally, the initiator MSS starts a fourth execution of the Echo protocol to inform the MSSs about the completion of the counting algorithm. After the completion of the fourth Echo protocol, an MSS stops to broadcast "size" messages when a new mobile host enters its cell.

The four executions of the Echo protocol imply that the algorithm requires $8|E|$ messages to be exchanged in the fixed part of the network yielding a total cost of $O(mC_w + |E|C_f) = O(mC_w + n^2C_f)$ which is better than the cost of the VTA algorithm.

The simulated implementations of the VTA and the TTA algorithms were done on the DSP tool. One of the major difficulties in the experimental setup was the modeling of the speed and the type of movement of a mobile host. When a mobile host moves fast, it will change many cells during the execution of any counting algorithm and thus increase the overhead of keeping the routing tables of MSSs updated. Since the description of the speed in terms of physics was rather difficult, the approach followed in [10.16] was to associate the speed of a host with the propagation delay of messages in the fixed part of the network. In these terms, a *slow* mobile host is a host which does not change cell for $O(D)$ time units (where D is the diameter of the fixed part of the network). A host which changes cell in time smaller than $O(D)$ is called a *fast* mobile host.

Five different topologies were considered in [10.16] with n (number of MSSs) ranging from 20 to 100, $|E|$ (number of edges in the fixed part of the network) ranging from 50 to 310, and diameter ranging from 5 to 21. The transmission delay in all links was unary in order to avoid the overhead of message delay in the protocol execution time. In all topologies, there were 10 mobile hosts in each cell of an MSS. At the beginning of the simulation the mobile hosts were left to move randomly in the network in order to take random positions before a counting algorithm starts. For the VTA algorithm, the virtual topology constructed was basically a list of mobile hosts where the host with identity i considered as its neighbors the hosts with identities $i-1$ and $i+1$.

Three parameters were measured in all experiments: (i) the number M_f of messages exchanged in the fixed part of the network in order to deliver messages to the mobile hosts; (ii) the number M_r of radio messages transmitted by the mobile hosts (excluding the "join" messages, since these are also used for network control and routing purposes); and (iii) the execution time of the protocol. The last two parameters express the *battery power* of a mobile host, since this power depends on the number of message transmissions made by the mobile host as well as on the time the host remains active (algorithm execution time).

The TTA algorithm was significantly better than VTA in any topology and for any measurement parameter considered both in battery consumption and in the load of the fixed network. For example, in topology 5 ($n = 100$, $|E| = 310$, $D = 21$) and for slow mobile hosts, M_f was 29350 for VTA and 2480 for TTA, M_r was 2000 for VTA and 1000 for TTA, while the execution time was 18152 for VTA and 171 for TTA. Very similar results hold for fast mobile hosts.

10.5.3 Basic Protocols in the Ad-Hoc Model

A fundamental problem in the ad-hoc model is to send a piece of information from some sender host to another designated receiver host. This *basic communication* or *routing* problem is a highly non-trivial task in ad-hoc mobile networks for several reasons: (a) local connections are temporary and may change as users move; (b) the movement rate of each user might vary, while certain hosts may even stop in order to execute location-oriented tasks.

The most common way to establish communication is to form paths of intermediate nodes (i.e., hosts), where it is assumed that there is a link between two nodes if the corresponding hosts lie within one another's transmission radius and hence can directly communicate with each other [10.14, 10.36, 10.43]. In other words, starting from the sender, each host broadcasts the message to all its neighbors until the intended receiver gets it (if possible). This protocol is called *flooding* and clearly requires a lot of messages. Indeed, this approach of exploiting pairwise communications is common in ad-hoc mobile networks that either cover a relatively small space (i.e., the temporary network has a small diameter with respect to the transmission range), or are dense (i.e., thousands of wireless nodes). Since almost all locations are occupied by some hosts, broadcasting can be efficiently accomplished.

In wider area ad-hoc networks however, broadcasting is impractical, as two distant hosts will not be reached by any broadcast since users do not occupy all intervening locations, that is, a sufficiently long communication path is difficult to establish. Even if such a path is established, single link failures happening when a small number of users that were part of the communication path move in a way such that they are no longer within the transmission range of each other, will make this path invalid. Note also that the path established in this way may be very long, even in the case of connecting nearby nodes.

A different approach to solve this basic communication problem is to take advantage of the mobile hosts natural movement by exchanging information whenever mobile hosts meet incidentally. Protocols based on this idea are divided into *non-compulsory* and *compulsory protocols*.

A *non-compulsory* protocol is one whose execution does not affect the movement of the mobile host. When the users of the network meet often and are spread in a geographical area, flooding the network will suffice. It is evident, however, that if the users are spread in remote areas and they do not move beyond these areas, then there is no way for information to reach them, unless the protocol takes care of such situations.

One way to alleviate these problems is to force mobile users to move according to a specific scheme in order to meet the protocol demands, thus yielding the so-called *compulsory* protocols. Such a protocol requests that *all* mobile hosts perform certain moves in order to guarantee correctness of the protocol.

A compromise between non-compulsory and compulsory protocols is introduced in [10.7]. The idea is to force only a small subset of mobile users,

called the *support* Σ of the network, to move as per the needs of the protocol (the move of the rest is arbitrary and is not affected by the protocol). Such protocols are called *semi-compulsory* protocols. The support serves as an intermediate pool for receiving and delivering messages.

To address the crucial issue of modeling the motions of mobile hosts in the three-dimensional space, a rather general graph theoretic model was introduced in [10.17]. Under this model, the space S of motions is mapped to a graph $G = (V, E)$ called the *motion graph*. The graph is constructed as follows. The space S is quantized in cubes. Each cube has a volume that approximates (from below) the volume of a sphere which represents the transmission range of a mobile host. The motion graph has a vertex for each cube of the quantization of S. Two vertices are connected by an edge if their corresponding cubes are adjacent. Note that the number of vertices n of G approximates the ratio of the volume of S and the space occupied by the transmission range of a mobile host. Since edges represent the (at most 6) neighboring polyhedra of a cube, it follows that $|E| = O(n)$. The mobile hosts move along the vertices and edges of the motion graph G (note that the motion graph model neglects the detailed geometric characteristics of the motion). It is assumed that the hosts know in advance (for example, from the hardware) the type and the dimensions of the polyhedron that is used for the quantization of S in order to be able to determine whether they have covered enough distance to reach a new vertex of G.

In the rest of this section, we shall discuss two efficient semi-compulsory protocols for the basic communication problem developed and implemented in [10.7, 10.8], and which model motions of hosts using the motion graph. Although "hard" input instances (in the sense of Section 10.4) were not considered, the experiments were conducted on several interesting "pragmatic" inputs.

10.5.3.1 The Snake Protocol. The first semi-compulsory protocol for the basic communication problem was presented in [10.7]. It uses a snake-like sequence of k support stations (i.e., they form a list of k nodes) that always remain pairwise adjacent and move in a way determined by the snake's head. As a consequence, the protocol is referred to as the *snake protocol*. There is a set-up phase of the ad-hoc network, during which a predefined number, k, of hosts, become the nodes or members of the support. The head is determined by performing a leader election between the members of Σ. Once determined, the head (denoted by M_0) assigns unique names to the rest of the support members $M_1, M_2, \ldots, M_{k-1}$. The motion of the support stations is accomplished in a distributed way via a support motion subprotocol P_1 which enforces the support to move as a "snake", with the head M_0 doing a random walk on the motion graph and each of the other nodes M_i executing the simple protocol "move where M_{i-1} was before". When some node of the support is within the communication range of a sender, an underlying sensor subprotocol P_2 notifies the sender that it may send its message(s). The mes-

sages are then stored in every node of the support using a synchronization subprotocol P_3. When a receiver comes within the communication range of a node of the support, the receiver is notified that a message is "waiting" for him and the message is then forwarded to the receiver. Duplicate copies of the message are then removed from the other members of the support. In this protocol, the support Σ plays the role of a (moving) backbone subnetwork (of a "fixed" structure, guaranteed by the motion subprotocol P_1), through which all communication is routed.

The snake protocol is theoretically analyzed in [10.7], where it is shown that the total expected communication or delay time to send a message from a sender to a receiver is at most $\frac{2}{\lambda_2(G)}\Theta(n/k) + \Theta(k)$ where G is the motion graph, $\lambda_2(G)$ is its second eigenvalue, n is the number of vertices in G, and $k = |\Sigma|$.

A first implementation of the protocol was developed and experimentally evaluated in [10.7] with the emphasis to confirm the theoretical analysis, and to investigate whether it is helpful for the head of Σ to remember past positions occupied by Σ, thus avoiding them in the future. The implementation was done in `C++` using LEDA [10.33].

The experimental setup in [10.7] consisted of three kinds of inputs, one random and two structured ones. Each kind of input corresponded to a different type of motion graph. The motion graphs considered were random graphs (a natural starting point), 2D grid graphs (the simplest model of motion when mobile hosts move on a plane surface), and bipartite multi-stage graphs. The latter type of graph consists of a number s of stages (or levels) where each stage consists of n/s vertices. There are edges between vertices of consecutive stages chosen randomly among all possible edges between the two stages. This type of graphs is interesting as such graphs model movements of hosts that have to pass through certain places or regions, and have a different second eigenvalue than grid and random graphs (their second eigenvalue lies between that of grid and random graphs).

For all these types of graphs several values for n in the range $[100, 6400]$ were considered and different values for the support size k in the range $[5, 40]$. For each motion graph constructed, $1,000$ users (mobile hosts not belonging to Σ) were injected at random positions that generated 100 transaction message exchanges of 1 packet each by randomly picking different destinations (i.e., a total of $100,000$ messages were transmitted). The move of each user was random and independent of the protocol. Each experiment was carried out until all $100,000$ messages were delivered to the designated receivers. The synchronization subprotocol P_3 (storing every message to each member of Σ) was not implemented and hence the extra delay imposed by this subprotocol was not counted in the measured delay times. This does not affect the behaviour of the snake protocol and helps simplifying the implementation.

The conducted experiments [10.7] indeed confirmed the theoretical analysis that only a small support is needed for efficient communication, and

indicated that limited memory (remembering just a few past positions) incurs a slight improvement on the delay time, while bigger memory is not helpful at all.

10.5.3.2 The Runners Protocol. The second semi-compulsory protocol for the basic communication problem has been recently presented in [10.8]. This protocol is based on the idea that the members of Σ do not to move in a snake-like fashion, but they perform *independent* random walks on the motion graph G, that is, the members of Σ can be viewed as "runners" running on G. In other words, instead of maintaining at all times pairwise adjacency between members of Σ, all hosts sweep the area by moving independently from each other. Consequently, this protocol is referred to as the *runners protocol*. When two runners meet, they exchange any information given to them by senders encountered using a new synchronization subprotocol P_3'. As in the snake case, when some node of the support is within the communication range of a sender, the underlying sensor subprotocol P_2 notifies the sender that it may send its message(s). When a user comes within the communication range of a node of the support which has a message for the designated receiver, the waiting messages are forwarded to the receiver. The runners protocol does not use the idea of a (moving) backbone subnetwork as no motion subprotocol P_1 is used. However, all communication is still routed through the support Σ and it is expected that the size k of the support (number of runners) will affect performance in a more efficient way than that of the snake approach. This expectation stems from the fact that each host will meet each other in parallel, accelerating the spread of information (that is, the messages to be delivered). A member of the support stores all undelivered messages in a set S_1, and maintains a list of receipts S_2 to be given to the originating senders. When two runners meet at the same site of the motion graph G, the synchronization subprotocol P_3' is activated. The subprotocol imposes that when runners meet on the same site, their sets S_1 and S_2 are synchronized. In this way, a message delivered by some runner will be removed from the set S_1 of the rest of runners encountered, and similarly delivery receipts already given will be discarded from the set S_2 of the rest of runners. The synchronization subprotocol P_3' is partially based on the *two-phase commit* algorithm as presented in [10.30].

The runners protocol turns out to be more robust than the snake protocol. The latter is resilient only to one fault (one faulty member of Σ), while the former is resilient to t faults for any $0 < t < k$.

In [10.8], a comparative experimental study of the snake and the runners protocols was conducted based on a new generic framework developed to implement protocols for mobile computing which constitutes part of the basic primitives provided by the Distributed Algorithms Platform [10.10] for wireless computing. Under this framework, the implementation of the runners protocol and the re-implementation of the snake protocol were carried out. All implementations were done in `C++` using LEDA [10.33] and the prim-

itives of DAP [10.10]. To provide a fair comparison between the two different protocols (snake and runners), the subprotocol P_3 of the snake protocol was implemented in [10.8], and consequently the extra delay imposed by the synchronization of the mobile support hosts was also counted.

The experimental setup in [10.7] has been extended in [10.8] to include more pragmatic test inputs regarding motion graphs. Hence, except for the test inputs considered in [10.7] (random graphs, 2D graphs, bipartite multistage graphs; see Section 10.5.3.1), two other structured families were considered: 3D graphs (modeling 3D space), and two-level graphs. The latter class consists of dense subgraphs interconnected by a small number of paths. It was motivated by the fact that most mobile users usually travel along favourite routes (e.g., going from home to work and back) that usually comprise a small portion of the whole area covered by the network (e.g., urban highways, ring roads, metro lines), and that in more congested areas there is a high volume of user traffic (e.g., city centers, airport hubs, tourist attractions). In the conducted experimental study, the primary interest was to provide measures on communication times (especially average message delay), message delivery rate, and support utilization (total number of messages contained in all members of the support).

The experiments in [10.8] revealed that: (i) for both protocols only a small support is required for efficient communication; (ii) the runners protocol outperformed the snake protocol in almost all types of inputs considered. More precisely, the runners protocol achieve a better average message delay in all test inputs considered, except for the case of random graphs with a small support size. The runners protocol achieves a higher delivery rate of messages right from the beginning, while the snake protocol requires some period of time until its delivery rate stabilizes to a value that is always smaller than that of runners. Finally, the runners protocol utilizes more efficiently the available resources as far as memory limitations are concerned, as it has smaller requirements for the size of local memory per member of the support.

10.6 Modeling Attacks in Networks: A Useful Interplay between Theory and Practice

A recent thread of research concerns attacks in computer networks which pose several key problems regarding intrusion propagation and detection. Various models have been proposed under which researchers mainly study the effective detection and defeat of attacks assuming a very powerful intruder; see for example, [10.26, 10.35]. In this setting, intrusion propagation (the process of spread of such attacks) has mostly been investigated under gossip or epidemiological models [10.26]. On the other hand, the fear of malicious attacks along with the development of advanced cryptographic techniques has considerably increased the security level of current computer systems. Hence,

contrary to previous models and approaches, a recent work [10.34] is concerned with studying intrusion propagation assuming that the intruder has a rather limited power and aims at investigating how intrusion can propagate in a perhaps highly secure network. To this end, a general model for such an intrusion and its propagation in networks is introduced. In the rest of this section, we shall discuss the particular model as well as the development of distributed protocols for attack propagation under this model. The interesting issue is that a tight combination of analytic and experimental methods is used to develop the protocols.

In the model introduced in [10.34], a network $\mathcal{N}$ is viewed as a collection of n host systems (nodes) each one having its own logical address. There is some underlying physical infrastructure whose specific topology is not a concern of the model. Communication is not necessarily done point-to-point. Direct communication between two nodes is achieved by establishing a virtual channel through the physical infrastructure between these two nodes.

Assume that in such a network an intruder, starting from his own computer, would like to break as many other systems as possible. The intrusion consists of a collection of attacks. An *attack* is issued from some node in $\mathcal{N}$ and is an attempt to break the perimetric security of another node (host system) in $\mathcal{N}$. The intrusion is realized by an attack scheme. An *attack scheme* is a protocol for the organization of the attacks issued from specific nodes of $\mathcal{N}$. The intruder is a greedy one, i.e., does not have a specific target, and attacks computer systems equiprobably at random. An attack succeeds or fails, independently of other attacks, with a failure probability $0 < f < 1$ that represents the difficulty of breaking a system in $\mathcal{N}$; f is a gross measure of the security level of the attacked systems (e.g., of the average security or the perceived maximum security level of a system) and may also depend on the intruder's skills. The model assumption about f is motivated by a large class of existing attacks; for example, attacks that are based on randomly sampling a set of possible passwords from a large password domain and then trying each of them. The probability of success of such a scheme in a node does not depend on previous successes at other nodes or on previous attempts at the same node. This is because the locally implemented set of passwords is perhaps different in each node and the set of passwords used by the local attack software is very small (for reasons of speed) compared to the password domain set.

If an attack does not fail, then some, randomly and equiprobably chosen, network node is returned. Because of that, it may happen that an already selected node (an already broken system) is chosen again. If the result of a non-failed attack is a node which has not been selected before, then the attack is considered *successful* and a *virtual link* (virtual channel) is established to that node. The random selection, with possible repetition, of a node in the case of a non-failed attack is motivated by the following pragmatic considerations: (i) if the local attack software (e.g., a worm) is successfully

confronted, it should not reveal any information about broken nodes in the past; (ii) the local attack software may blindly extend attacks to hosts contained in tables of the newly broken systems which may include the already broken ones. The intruder tries to protect himself as much as possible from being traced: once a system is broken, his software tries from *that* system to attack (again equiprobably at random) another system by disguising itself as a user process of the broken system. Because of the danger to be discovered, the intruder's software can ever try only a limited (i.e., constant) number g of attacks from a specific node of the network. If a successful attack is issued before the limit g is reached, then the software enters a dormant phase and performs no action (for the purpose of not raising any suspicions). If at some node i the software exhausts the attack bound g, then it terminates execution at i and "backtracks" to a previously broken system j to continue from there its attacks, provided that there are still some attempts left at j. In such a case, the local software at j is reactivated and starts again to issue attacks. If at any time during the execution of the attack scheme, the intrusion is discovered by some system, it is assumed that the whole attack scheme to $\mathcal{N}$ terminates.

Two natural questions raised here are: (a) how long the intruder can go on (i.e., how many computer systems can be successfully attacked) in $\mathcal{N}$ until he is discovered, and (b) how many virtual links a detection mechanism has to trace in order to find the origin of the intrusion. In particular, assume that the intrusion starts at time 0 with attack scheme S. At any time $t \geq 0$, let $n_S(t)$ be the number of nodes captured, called the *spread factor*, and let $\ell_S(t)$ be the shortest possible distance (in number of virtual links) from the currently active position of the intruder's software to the origin, called the *traceability factor*. Given a discovery (i.e., stopping) time T, the problem is to estimate $n_S(T)$ and $\ell_S(T)$. This is referred to as the *attack propagation* problem. The goal, from the side of the intruder, is to employ an attack scheme which maximizes *both* factors. Note that this is a non-trivial task; for example, almost all epidemiological (and gossip propagation) models have usually very small $\ell_S(T)$ compared to $n_S(T)$, because of their "radially" spreading nature.

The above process defines naturally a graph G whose vertices correspond to the nodes of the network and if a virtual link (i.e., a successful attack through some virtual channel) is established between two nodes i and j, then an edge between vertices i and j is added to G. In this setting, the spread factor $n_S(T)$ is the size of the obtained connected component in G, and the traceability factor $\ell_S(T)$ is the length (number of edges) of the path in this connected component from the current vertex (node) issuing attacks to the origin (the node from which the intrusion was started). This path is referred to as the *traceability path*. Hence, the attack propagation problem reduces in estimating the values of these two quantities in G.

Another interesting issue is to investigate the possibility of a total failure of an attack scheme, namely the possibility that it eventually returns to its starting point, not because the intruder is discovered but due to backtracking caused by the limited number of attempts from a specific node.

In [10.34], the attack propagation problem is tackled by presenting four different protocols (attack schemes) by which an intruder can organize his attacks in the above model. The starting point is an attack propagation protocol that organizes attacks along a single traceability path. The g attacks per node are grouped into three equally sized sets. The first two sets (called *green* and *red*, resp.) are used to propagate the attack, while the third set is kept for restarting the attack scheme in case of a total failure. The protocol tries initially to establish a (long) traceability path, link by link, using only the green attacks. Each attack is issued from the last node in the path which is considered active (i.e., it possesses a token). An attack is considered successful if a new node is returned which will now become the last node of the path and gets the token. When attack propagation, that is, extension of the constructed path using green attacks, is not possible, then the red set of attacks is used. If extension to a new node is established, then the protocol passes the token to that node and continues from there using its green attacks. Otherwise, it backtracks to the first node whose red attacks have not been used yet and (after passing the token) tries to extend the path from that node using the red attacks. The node having the token always stores the maximum (in length) traceability path of attacked systems constructed so far by the protocol. When the path shrinks to a single node, then that node notifies all nodes of the maximum traceability path seen in the past to try to use their third batch of attacks in order to restart the protocol. The above protocol is referred to as the *original protocol* and forms the basis for the development of three other protocols, called *tree protocols*.

The tree protocols are based on the fact that the graph G, constructed incrementally during the execution of the original protocol, is actually a tree (only successful attacks are recorded as edges of G). Hence, instead of keeping only the maximum traceability path constructed, the idea is to store the whole tree. Subsequently, various orders of the nodes in this tree for path extension are considered using their third batch of attacks. The different orders specify the different ways the intruder can use to organize his attacks. Clearly, the maximum traceability path constructed by the original protocol is the path of maximum depth in the tree. Hence, the tree protocols provide naturally a bigger front for expansion. The tree protocols differ in the order the nodes of the tree are considered for path expansion. Two protocols are based on "reverse" DFS, while the third one is based on "reverse" BFS.

The above protocols are theoretically analyzed in [10.34] where also an implementation of the protocols was carried out (in a simulation environment) along with a comparative experimental study. The development of the protocols constitutes an interesting case of distributed algorithm engineering.

For the analysis of the protocols, both analytic and experimental methods were used that are actually tied to each other. The analytic study of the protocols, where applicable, was rather complicated and gave only lower bounds that are (probably) not tight. Hence, resorting to experiments was the only way to get insight as well as a basis of reasonable assumptions to further proceed with the analysis. For example, it was crucial in the analysis of the tree protocols to find a lower bound on the ratio between the traceability factor and the size of the tree. The experiments clearly demonstrated a lower bound of $1/2$ for this ratio which, along with the tree evolution observed experimentally, helped to analytically prove it and complete the analysis. Moreover, the implementation and experimentation with the original protocol provided useful feedback which was crucial in the development of the tree protocols.

The analytic and experimental methods in [10.34] show that for *any* $0 < f < 1$, there exists a g for which any of the above attack schemes will achieve a $\Theta(n)$ spread factor with high probability, provided T is sufficiently large. This means that if an intrusion is realized by any of the attack schemes, it will spread, regardless of the security level, to a big part of the network. It is also shown that the spread and the traceability factors are linearly related. Actually, for the tree protocols this linear relationship holds, with high probability, during the *whole* duration of the attack propagation. This implies that it will not be easy for a detection mechanism to trace the origin of the intruder, since at any time it will have to trace a number of links proportional to the number of nodes captured. Finally, it is shown that the probability of a total failure of any attack scheme is very small. The experiments conducted in [10.34] verified the theoretical results and exhibited the robustness of one tree protocol.

10.7 Conclusion

Distributed algorithm engineering has certain characteristics which are very different from conventional algorithm engineering. In this work, we made a first attempt to address these issues and suggest possible approaches that could efficiently tackle them. We hope that the suggested approaches will inspire researchers to further investigate these issues and result in more systematic methods.

References

10.1 A. Acharya, B. Badrinath, and T. Imielinski. Structuring distributed algorithms for mobile hosts. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994.

10.2 M. Adler, H. Räcke, N. Sivadasan, C. Sohler, and B. Vöcking. Randomized pursuit-evasion in graphs. In *Automata, Languages, and Programming (ICALP'02)*. Springer Lecture Notes in Computer Science, to appear.

10.3 H. Attiya and J. Welch. *Distributed Computing*. McGraw-Hill, 1998.

10.4 B. Awerbuch and D. Peleg. Concurrent online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, 1995.
10.5 M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–863, 1984.
10.6 J. Buss, P. Kanellakis, P. Ragde, and A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20:45–86, 1996.
10.7 I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis. Analysis and experimental evaluation of an innovative and efficient routing approach for ad-hoc mobile networks. In *Proceedings of the 4th Workshop on Algorithmic Engineering (WAE'00)*. Springer Lecture Notes in Computer Science 1982, pages 99–110, 2000.
10.8 I. Chatzigiannakis, S. Nikoletseas, N. Paspallis, P. Spirakis, and C. Zaroliagis. An experimental study of basic communication protocols in ad-hoc mobile networks. In *Proceedings of the 5th Workshop on Algorithmic Engineering (WAE'01)*. Springer Lecture Notes in Computer Science 2141, pages 159–171, 2001.
10.9 O. Dahl, K. Nygaard. SIMULA — an Algol-based simulation language. *Communications of the ACM*, 9(9):671-678, 1966.
10.10 Distributed Algorithms Platform. `http://ru1.cti.gr/~LEP-DAP/`.
10.11 Distributed Systems Platform. `http://helios.cti.gr/alcom-it/dsp`.
10.12 The Esterel language. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.
10.13 M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
10.14 Z. Haas and M. Pearlman. The performance of a new routing protocol for the reconfigurable wireless networks. In *Proceedings of ICC'98*, 1998.
10.15 K. Hatzis, G. Pentaris, P. Spirakis, and V. Tampakas. Implementation and testing eavesdropper protocols using the DSP tool. In K. Mehlhorn, editor, *Proceedings of the 2nd Workshop on Algorithmic Engineering (WAE'98)*, pages 74–85, 1998.
10.16 K. Hatzis, G. Pentaris, P. Spirakis, and V. Tampakas. Counting in mobile networks: theory and experimentation. In *Proceedings of the 3rd Workshop on Algorithmic Engineering (WAE'99)*. Springer Lecture Notes in Computer Science 1668, pages 95–109, 1999.
10.17 K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas, and R. Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 251–260, 1999.
10.18 M. Herlihy and S. Rajsbaum. Algebraic topology and distributed computing: a primer. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer Lecture Notes in Computer Science 1000, pages 203–217, 1995.
10.19 T. Imielinski and H. F. Korth. *Mobile Computing*. Kluwer Academic Publishers, 1996.
10.20 The IOA homepage. `http://theory.lcs.mit.edu/tds/ioa.html`.
10.21 J. R. Jump. *NETSIM Reference Manual, Version 1.0*. Rice University, 1993.
10.22 J. R. Jump. *YACSIM Reference Manual, Version 2.1*. Rice University, 1993.
10.23 P. Kanellakis and A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.
10.24 Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proceedings of the 23rd ACM Symposium on Theory of Computing (STOC'91)*, pages 381–390, 1991.

10.25 Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC'90)*, pages 138–148, 1990.
10.26 J. Kephart and S. White. Directed-graph epidemiological models of computer viruses. IBM Research Report; also, in *Proceedings of the IEEE Symposium on Security and Privacy*, 1991.
10.27 E. Koutsoupias and C. Papadimitriou. Worst-case equilibria. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*. Springer Lecture Notes in Computer Science 1563, pages 404–413, 1999.
10.28 L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-564, 1978.
10.29 M. C. Little and D. McCue. Construction and use of a simulation package in C++. *C User's Journal*, 12(3), 1994. Also at `http://cxxsim.ncl.ac.uk/`.
10.30 N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.
10.31 M. Mavronicolas and P. Spirakis. The price of selfish routing. In *Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC'01)*, pages 510–519, 2001.
10.32 S. McCanne and S. Floyd. *ns Network Simulator*. `http://www.isi.edu/nsnam/ns/`.
10.33 K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
10.34 S. Nikoletseas, G. Prasinos, P. Spirakis, and C. Zaroliagis. Attack propagation in networks. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, pages 67–76, 2001. To appear in *Theory of Computing Systems*.
10.35 R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 51–59, 1991.
10.36 V. Park and M. Corson. Temporally-ordered routing algorithms (TORA): version 1 - functional specification. IETF, Internet Draft, draft-ietf-manet-tora-spec-02.txt, Oct. 1999.
10.37 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228-234, 1980.
10.38 C.E. Perkins. *Ad-Hoc Networking*. Addisson-Wesley, 2001.
10.39 P. Spirakis and B. Tampakas. Distributed pursuit-evasion: some aspects of privacy and security in distributed computing. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, short paper, 1994.
10.40 P. Spirakis, B. Tampakas, and H. Antonopoulou. Distributed protocols against mobile eavesdroppers. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95)*. Springer Lecture Notes in Computer Science 972, pages 160–167, 1995.
10.41 G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.
10.42 B. Weiss, G. Gridling, U. Schmid, and K. Schossmaier. The SimUTC fault-tolerant distributed systems simulation toolkit. In *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, 1999.
10.43 Y. Zhang and W. Lee. Intrusion detection in wireless ad-hoc Networks. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing (MOBICOM'00)*, pages 275–283, 2000.

11. Implementations and Experimental Studies of Dynamic Graph Algorithms

Christos D. Zaroliagis

[1] Computer Technology Institute
P.O. Box 1122, 26110 Patras, Greece

[2] Department of Computer Engineering & Informatics
University of Patras, 26500 Patras, Greece
zaro@ceid.upatras.gr

Summary.

Dynamic graph algorithms have been extensively studied in the last two decades due to their wide applicability in many contexts. Recently, several implementations and experimental studies have been conducted investigating the practical merits of fundamental techniques and algorithms. In most cases, these algorithms required sophisticated engineering and fine-tuning to be turned into efficient implementations. In this paper, we survey several implementations along with their experimental studies for dynamic problems on undirected and directed graphs. The former case includes dynamic connectivity, dynamic minimum spanning trees, and the sparsification technique. The latter case includes dynamic transitive closure and dynamic shortest paths. We also discuss the design and implementation of a software library for dynamic graph algorithms.

11.1 Introduction

The traditional design of graph algorithms usually deals with the development of an algorithm that, given a static (fixed) graph G as input, solves a particular problem on G; for example, "is G connected?". A *dynamic graph*, on the contrary, is a graph which may evolve with time due to local changes that occur in G; e.g., insertion of a new edge or deletion of an edge. The challenge for an algorithm dealing with a dynamic graph is to maintain, in an environment of dynamic local changes, a desired graph property (e.g., connectivity) efficiently; that is, without recomputing everything from scratch after a dynamic change. Dynamic graphs are usually more accurate models than static graphs, since most real systems (e.g., physical networks) are not truly static.

A *dynamic algorithm* is a data structure that allows two types of operations: queries and updates. A query asks for a certain property P of the current graph G (e.g., "are vertices x and y connected in G?"), while an update operation reflects a local change in G. Typical changes include insertion of a new edge and deletion of an existing edge. An algorithm or a problem is called *fully dynamic* if both edge insertions and deletions are allowed, and it is called *partially dynamic* if either edge insertions or edge deletions are allowed

R. Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 229–278, 2002.
© Springer-Verlag Berlin Heidelberg 2002

(but not both). In the case of edge insertions (resp. deletions), the partially dynamic algorithm or problem is called *incremental* (resp. *decremental*).

The main goal of a dynamic algorithm is to use structural properties of the current graph G in order to handle updates efficiently, i.e., without resorting to the trivial approach of recomputing P from scratch using a static algorithm. In most cases, updates take more time than queries, and the sequence of operations (updates and queries) is provided in an on-line fashion (i.e., the operations are not known in advance).

Dynamic graph algorithms have been an active and blossoming field over the last years due to their wide applicability in a variety of contexts, and a number of important theoretical results have been obtained for both fully and partially dynamic graph problems. These results show a clear distinction between problem solving in undirected and in directed graphs: maintaining a property (e.g., connectivity) in a directed graph turns out to be a much more difficult task than maintaining the same property on an undirected graph. There is a bulk of striking results and novel techniques for undirected graphs which cannot however be transferred to directed graphs.

The challenge for dynamic algorithms to beat their (usually very efficient) static counterparts as well as the fact that their input is more complicated than the input of the corresponding static algorithms, has sometimes led to the development of rather sophisticated techniques and data structures. This, however, makes their practical assessment a non-trivial task, since the actual running times may depend on several parameters that have to do with the size and type of input, the distribution of operations, the length of the operation sequence, the update pattern in the operation sequence, and others.

Hence, it is inevitable to perform a series of experiments with several dynamic algorithms in order to be able to select the most appropriate one for a specific application. On the one hand, this experimentation often requires sophisticated engineering and fine-tuning to turn theoretically efficient algorithms into efficient implementations. On the other hand, the conducted experiments give useful insight which can be used to further improve the algorithms and the implementations.

Experimentation, however, requires proper selection of the test sets on which the implemented dynamic algorithms will be assessed, i.e, the test set should be as complete as possible. This in turn implies that both unstructured (i.e., random) and structured inputs should be considered. The former is important to either confirm the average-case analysis of an algorithm, or (if such an analysis does not exist) to understand its average-case performance. The latter is equally important as it either provides more pragmatic inputs (inputs originated from or motivated by real-world applications), or provides worst-case inputs, that is, inputs which will enforce an algorithm to exhibit its worst-case performance. Random inputs are usually easier to generate than structured inputs, while generation of worst-case inputs is perhaps the most difficult as it depends on several factors (problem, algorithm, etc).

In this paper, we survey several implementations along with their experimental studies for dynamic problems on undirected and directed graphs. The former case includes dynamic connectivity, dynamic minimum spanning trees, and the sparsification technique. The latter case includes dynamic transitive closure and dynamic shortest paths. We also discuss the design and implementation of a software library for dynamic graph algorithms. All but one of the implementations have been done in `C++` using the LEDA platform for combinatorial and geometric computing [11.52].

To give a better picture on how the implementations stand in relation with the algorithms known from theory, the treatment of each dynamic problem starts by presenting first the known theoretical results and then discussing the available implementations, commenting on the data sets used, and concluding with lessons learned.

11.2 Dynamic Algorithms for Undirected Graphs

The implementation studies known for dynamic problems on undirected graphs concern dynamic connectivity and minimum spanning tree. For the rest of this section, $G = (V, E)$ represents an undirected graph with n vertices and m edges, unless stated otherwise.

11.2.1 Dynamic Connectivity

11.2.1.1 Theoretical Background — Problem and History of Results. In the dynamic connectivity problem, we are interested in answering *connectivity queries* in a graph G which undergoes a sequence of updates (edge insertions and edge deletions). Given any two vertices x and y, a connectivity query asks whether there is a path in G between x and y. The dynamic connectivity problem reduces to the problem of maintaining a spanning forest in G, i.e., maintaining a spanning tree for each connected component of G. Dynamic connectivity was studied both in a fully and in a partially dynamic setting.

The first algorithm for fully dynamic connectivity was given by Harel [11.35]; it supported queries in $O(1)$ time and updates in $O(n \log n)$ time. Frederickson [11.25] reduced this update bound to $O(\sqrt{m})$. This was later improved by Eppstein et al. [11.19] to $O(\sqrt{n})$ through the use of a very simple but powerful technique called sparsification, which is a general method for producing dynamic algorithms. Further improvements came with the use of randomization. The first such algorithm (of Las-Vegas type) was presented by Henzinger & King [11.36] achieving $O(\log^3 n)$ expected amortized time for updates and $O(\log n / \log \log n)$ time for queries. The expected amortized update time was subsequently improved to $O(\log^2 n)$ by Henzinger & Thorup [11.39]. At about the same time, Nikoletseas et al. [11.55] presented a fully dynamic,

probabilistic (Monte-Carlo), connectivity algorithm for random graphs and random update sequences which also achieves $O(\log^3 n)$ expected amortized time for updates, but answers queries in $O(1)$ expected amortized time. The need for randomization was removed by Holm et al. [11.40]; in that paper a deterministic algorithm for fully dynamic connectivity is presented which achieves $O(\log^2 n)$ update time and $O(\log n/\log\log n)$ query time. Very recently Thorup [11.65] presented a new randomized (Las-Vegas) fully dynamic algorithm with $O(\log n(\log\log n)^3)$ expected amortized time for updates and $O(\log n/\log\log\log n)$ time for queries. It is worth noting that the above polylogarithmic upper bounds for updates and queries are not far away from the currently best lower bound of $\Omega(\log n/\log\log n)$ [11.27, 11.53] for both operations. All the above algorithms with polylogarithmic update and query time require $O(m + n\log n)$ preprocessing time and space. Thorup also showed in [11.65] that the space bound of the algorithms in [11.40, 11.65] can be reduced to $O(m)$.

For partially dynamic connectivity, there are only two algorithms which achieve better results than those provided by the fully dynamic ones: an incremental algorithm based on Tarjan's union-find data structure [11.62] which achieves an amortized time of $O(\alpha(m, n))$ per update or query operation; a decremental randomized algorithm due to Thorup [11.64] which supports queries in $O(1)$ time and any number of edge deletions in a total $O(\min\{n^2, m\log n\} + \sqrt{nm}\log^{2.5} n)$ expected time, where m denotes the initial number of edges. This is $O(1)$ amortized expected time per operation if $m = \Omega(n^2)$.

11.2.1.2 Implementations and Experimental Studies. There are three works known regarding implementation and experimental studies of dynamic connectivity algorithms. In chronological order, these are the works by Alberts et al. [11.3], Fatourou et al. [11.22], and Iyer et al. [11.44].

The first paper investigates the practicality of sparsification-based approaches and their comparison to the randomization-based approach by Henzinger & King [11.36]. The second paper enhances this study by investigating the comparison between two randomized approaches, the one by Henzinger & King [11.36] and the other one by Nikoletseas et al. [11.55]. Finally, the third paper brings the algorithm by Holm et al. [11.40] into play and aims at investigating in practice the difference in the logarithmic improvement over the algorithm of [11.36]. Moreover, that study considerably enhances the data sets used in the experiments.

11.2.1.2.a *The Implementation by Alberts et al.* [11.3]

The main goal of the first experimental study for dynamic connectivity algorithms was threefold:

1. To investigate the practicality of dynamic algorithms over static ones (especially to very simple and easily implementable static algorithms).
2. To investigate the practicality of the sparsification technique and confirm the theoretical analysis regarding its average-case performance.

3. To compare dynamic algorithms based on sparsification with other dynamic algorithms and especially with the algorithm by Henzinger & King [11.36] which is based on randomization.

Sparsification is a simple and elegant technique which applies to a variety of dynamic graph problems. It can be used either on top of a static algorithm in order to produce a dynamic one, or on top of a dynamic algorithm in order to speed it up. Sparsification works as follows. The edges of G are partitioned into $\lceil m/n \rceil$ sparse subgraphs, called *groups*, each one having n edges. The remaining group, called the *small group*, contains between 1 and n edges. The information relevant for each subgraph (e.g., connectivity) is summarized in an even sparser subgraph called a *sparse certificate* (e.g., a spanning forest). In a next step certificates are merged in pairs yielding larger subgraphs which are made sparse by computing again their certificate. This step is applied recursively resulting in a balanced binary tree, called *sparsification tree*, in which each node represents a sparse certificate. Since there are $\lceil m/n \rceil$ leaves, the sparsification tree has height $\lceil \log(m/n) \rceil$. When an edge is inserted, it is placed in the small group; if there are already n edges in this group, then a new small group is created. When an edge is deleted, it is removed from the group to which it belongs and an edge from the small group is moved to the group which contained the deleted edge. If the last edge of the small group is deleted, the small group is removed. Consequently, an update operation (edge insertion/deletion) involves some changes to a $O(1)$ number of groups plus the examination of the sparse certificates (ancestors of the modified groups) in a $O(1)$ number of leaf-to-root tree paths. This in turn implies the examination of $O(\log(m/n))$ subgraphs of $O(n)$ edges each, instead of considering one large graph with m edges. This immediately speeds up an $f(n, m)$ time bound (representing either the time of a static algorithm or the update bound of a dynamic algorithm) to $O(f(n, O(n)) \log(m/n))$ and is called *simple sparsification*. The logarithmic factor of the previous bound can be eliminated with the use of more sophisticated graph decomposition and data structures resulting in the so-called *improved sparsification* (see [11.19] for the details).

Simple sparsification comes into three variants, depending on whether the certificates are recomputed by a static, fully dynamic, or partially dynamic algorithm. We shall keep the term *simple sparsification* for the first and third variants, since the second variant requires that certificates obey a so-called stability property and hence it is referred to as *stable sparsification* .

The simple sparsification was implemented in [11.3]. To achieve better running times, a few changes w.r.t. the original algorithm were introduced in the implementation:

(i) A queue keeps track of edge deletions in the groups. Namely, when an edge is deleted from a group, a pointer to that group is inserted in the queue (i.e., the queue represents "empty slots" in groups). When an edge is inserted, the first item is popped from the queue and the edge

is inserted into the corresponding group. If the queue is empty, then the new edge is inserted as in the original algorithm (i.e., either in the small group, or in a new small group). As a consequence, the deletion of an edge needs no swapping of edges and involves the examination of only one leaf-to-root path. According to the experiments in [11.3], this modification yields roughly 100% speedup for edge deletions.

(ii) The above implementation may however impoverish the edge groups: an update sequence with less insertions than deletions may invalidate the group size invariant (i.e., some group may have less than n edges) and result in a sparsification tree with height larger than $\lceil \log(m/n) \rceil$. To confront this situation, the sparsification tree is rebuilt each time its actual height differs by more than one from its "correct" height of $\lceil \log(m/n) \rceil$.

(iii) During an update, not all certificates in the leaf-to-root path are recomputed. Recomputation stops at the first tree node whose certificate remains unaffected by the update, since all its ancestors will not be affected as well. This introduces significant time savings on the average, and it is also matched by a theoretical analysis [11.3] which shows that on the average the number of sparsification nodes affected by an update is bounded by a small constant.

For the dynamic connectivity problem, simple sparsification was implemented on top of an incremental algorithm (based on the `Spanning_Tree` function of LEDA) which supports edge insertions in $O(\alpha(m,n))$ time and recomputes the solution from scratch after an edge deletion in $O(n+m\alpha(m,n))$ time. This results in an update time of $O(n\alpha(n,n)\log(m/n))$. The resulting implementation is called `Sparsification`.

The second algorithm implemented in [11.3] was the fully dynamic algorithm of Henzinger & King [11.36], henceforth the HK algorithm. The algorithm maintains a spanning forest F of the current graph G. Connectivity queries are answered by checking whether two given vertices belong to the same tree of the forest. As edges are inserted or deleted, the forest is maintained so that it is kept spanning. Hence, a data structure is required which performs efficiently the operations of joining two trees by an edge, splitting a tree by deleting an edge, and checking whether two vertices belong to the same tree. In [11.36] a data structure called *Euler-tour trees* (ET-trees) is introduced for this purpose. An ET-tree is a standard balanced binary tree over the Euler tour of a tree and supports all the above operations in $O(\log n)$ time. The key idea is that when trees are cut or linked, the new Euler tours can be constructed by at most 2 splits and 2 concatenations of the original Euler tours, while rebalancing of ET-trees affects only $O(\log n)$ nodes.

Maintaining the spanning forest F of G using the ET-trees yields immediately a very efficient way to handle connectivity queries (obvious) and edge insertions: when an edge is inserted, check whether its endpoints belong to the same tree of the forest. If yes, do nothing; otherwise, insert it in

the forest by simply joining the two ET-trees it connects. Both operations can be accomplished in $O(\log n)$ time. Edge deletions, however, require some additional care. If the deleted edge e is not a tree edge, then it is simply discarded. Otherwise, it causes the tree T to which it belongs to split into two other trees T_1 and T_2. In order to maintain the invariant that the forest is spanning, we have to check whether there is a non-tree edge (an edge which does not belong to any tree) that rejoins T_1 and T_2. Such an edge, if exists, is called *replacement edge*. Consequently, a dynamic connectivity algorithm must find replacement edges quickly. Henzinger & King [11.36] use two nice ideas to achieve this. The first is to use random sampling among the (possibly many) non-tree edges incident to T. However, the set of edges that rejoin T, called the candidate set, may be a small fraction of the non-tree edges adjacent to T and hence it is unlikely to find a replacement edge for e among the sampled ones. Since examining all non-tree edges adjacent to T is undesirable, another approach is required to deal with such a situation. Here comes the second idea of [11.36]: maintain a partition of the edges of G into $O(\log n)$ levels, forming $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$ of G, $1 \leq i \leq l = O(\log n)$. The partition is done in a way such that edges in highly-connected parts of the graph are on upper levels while edges in loosely-connected parts are at lower levels[1]. For each level i, a spanning forest F_i is maintained for the graph whose edges are in levels $j \geq i$. If a tree edge e at level i is deleted, then the non-tree edges in the smaller subtree, say T_1, are sampled. If within $O(\log^2 n)$ samples a replacement edge is found, we are done. Otherwise, the cut defined by the deletion of e is too sparse for level i (i.e., the vast majority of the non-tree edges incident on T_1 have both endpoints in T_1). In such a case, all edges crossing the cut are copied to level $i - 1$ and the procedure is applied recursively on level $i - 1$. Since edge insertions can cause the number of levels to increase beyond $O(\log n)$, the HK algorithm periodically rebuilds its data structure such that there are always $O(\log n)$ levels. The implementation of the above algorithm in [11.3] is referred to as `HK`.

A simplified version of the HK algorithm was also implemented in [11.3] and is referred to as `HK-var`. This version was motivated by experiments with random inputs which showed that it is very unlikely that edges move to lower levels. Hence, in the simplified version of the HK algorithm there is only one level and only $O(\log n)$ edges – instead of $O(\log^2 n)$ – are sampled. In this version, queries, edge insertions, and non-tree edge deletions still take $O(\log n)$ time, but the deletion of a tree edge may take $O(m \log n)$ worst-case time. Despite the latter, the simplified version was always faster than the original algorithm on random inputs and was more robust to input variations.

Finally, two pseudo-dynamic algorithms were implemented to provide a point of reference in the sense that they are the simplest possible methods

[1] The level notation here is the inverted version of the original algorithm in [11.36], in order to facilitate comparison with the forthcoming algorithm of [11.40].

one could resort to, using static approaches. Since these algorithms require only a few lines of code, their constants are expected to be rather low and hence likely to be fast in practice for reasonable inputs. These two algorithms are called `fast-update` and `fast-query`. The former spends only $O(1)$ time on updates (just updates two adjacency lists), but answers queries in $O(n+m)$ time using a BFS algorithm. The latter maintains a spanning forest and component labels at the vertices. Hence, a query takes $O(1)$ time (just checks equality of component labels). An update operation may force the current forest to change in which case the forest and the component labels are recomputed from scratch taking $O(n+m)$ time.

All the above implementations, `Sparsification`, `HK`, `HK-var`, `fast-update` and `fast-query` were compared experimentally in [11.3] on various types and sizes of graph inputs and operation sequences (updates intermixed with queries). Experiments were run both on random inputs (random graphs and operation sequences) as well as on non-random inputs (non-random graphs and operation sequences) representing worst-case inputs for the dynamic algorithms.

Random inputs are particularly important in the study of [11.3]. Recall that one of the main goals was to investigate the average-case performance of sparsification (and of the other algorithms), since in [11.3] the average-case running time of simple sparsification is analyzed and it is proved that the logarithmic overhead vanishes (the number of nodes affected by an update in the sparsification tree is bounded by a constant). The random inputs consisted of random graphs with different edge densities ($m \in \{n/2, n, n\ln n, n^{1.5}, n^2/4\}$). Note that the first three values constitute points where (according to random graph theory [11.9]) a radically different structural behaviour of the graph occurs: if $m \approx n\ln n$, the graph is connected (with high probability); if $n < m < n\ln n$, the graph is disconnected, has a so-called giant component of size $\Theta(n)$, and smaller components of size $O(\ln n)$ at most; if $m \approx n$, then the giant component has size $\Theta(n^{2/3})$; and if $m < n$, then the largest component has size $O(\ln n)$. The update sequences consisted of an equal number of m insertions, m deletions, and m queries, each one uniformly distributed on the candidate set. The candidate set for deletions was the set of current edges, for insertions the set of current non-edges w.r.t. the set of all possible edges, and for queries the set of all vertex pairs.

Non-random inputs aim at establishing a benchmark for inputs that could force dynamic connectivity algorithms to exhibit their worst-case performance. The non-random inputs consisted of structured graphs and operation sequences. A structured graph consists of a number k of cliques, each one containing roughly n/k vertices, and which are interconnected by $k-1$ inter-clique edges, called "bridges". The dynamic operations are only (random) insertions and deletions of bridges. As bridges are tree edges, constant insertion and deletion of them will cause the algorithms to constantly look for

replacement edges. Clearly, this kind of input represents a worst-case input for dynamic connectivity algorithms.

The first issue investigated was whether the theoretical analysis for the average-case performance of simple sparsification is confirmed in practice and whether `Sparsification` is better than the static algorithm on top of which it runs. The latter turned out to be true. The former was true for unweighted graphs (i.e., for problems like dynamic connectivity), but in the case of weighted graphs (e.g., for problems like dynamic minimum spanning tree) the experimental results did not always comply with the theoretical analysis, implying that perhaps a different model of analysis is required for such a case.

Regarding the comparison among the dynamic algorithms and the simple (pseudo-dynamic) ones, the reported experiments were as follows. For random inputs, `HK-var` was the fastest (as expected from the theoretical analysis), except for very sparse graphs ($m < n$) where `fast-query` was better. For non-random inputs, `Sparsification` and `HK` were better than the other algorithms; for large sequences of updates `HK` is faster, while for shorter sequences `Sparsification` is faster (the larger the update sequence, the better becomes the amortization in the HK algorithm). The behaviour of sparsification is due to the fact that it spreads the connectivity information in a logarithmic number of small subgraphs that have to be updated even if a local change does not affect the connectivity of the graph, i.e., tree and non-tree edge deletions produce roughly the same overhead. This turns out to be advantageous in the case of non-random graphs.

Another major conclusion of the study in [11.3] was that both sparsification and the HK algorithm proved to be really practical as they compare favorably to the simple algorithms even in the case of very small graphs (e.g., graphs with 10 vertices and 5 edges).

The source code of the above implementations is available from `http://www.jea.acm.org/1997/AlbertsDynamic`.

11.2.1.2.b *The Implementation by Fatourou et al.* [11.22]

The main goal of that study was to compare in practice the average-case performance of the fully dynamic, probabilistic algorithm by Nikoletseas et al. [11.55], henceforth NRSY, with the HK algorithm that appears to have a similar update bound and was also (along with the `fast-query`) among the fastest implementations for random inputs in the previous study [11.3].

The NRSY algorithm is different from the HK algorithm. It alternates between two epochs, the *activation* epoch and the *retirement* epoch, while it periodically performs *total reconstructions*, i.e., it rebuilds its data structure from scratch. A total reconstruction is called successful if it achieves in finding a giant component of size $\Omega(n)$ of the input random graph. The algorithm starts with a total reconstruction and builds a spanning forest of the graph. An activation epoch starts after a successful total reconstruction and ends when an edge deletion disconnects a spanning tree of the giant component

and the attempted fast reconnection fails. A retirement epoch starts either after an unsuccessful total reconstruction, or after the end of an activation epoch.

An *activation epoch* maintains a spanning forest of the graph and partitions the edges into three categories: *tree* edges, *retired* edges, and *pool* edges (which are reactivated retired edges). The activation epoch is divided into edge deletion intervals each consisting of $O(\log n)$ deletions. All edges marked retired during an edge deletion interval are re-marked as *pool* edges after the end of the next interval. An edge insertion during an activation epoch is performed as follows. If the inserted edge joins vertices in the same tree, then it is marked as *retired* and the appropriate data structures are updated; otherwise, the edge joins vertices of different trees and the component name of the smaller tree is updated. Edge deletions during an activation epoch are handled as follows. If the deleted edge is a pool or a retired edge, then it is simply deleted from all data structures it belongs. If the deleted edge is a tree edge, then we distinguish between two cases depending on whether this was an edge of the giant component or not. In the latter case, we look for a replacement edge and if the search is not successful, the tree is split and the smaller of the two resulted trees is relabeled. In the former case, a special procedure, called NeighborhoodSearch, is applied which performs two breadth-first searches (one in each tree) in tandem in an attempt to reconnect the tree of the giant component. The breadth-first searches (BFSs) stop either when the tree is reconnected or as soon as $O(\log n)$ vertices have been visited. NeighborhoodSearch proceeds in phases, where each phase starts when a vertex is visited during BFS. During a phase, an attempt is made to find a replacement edge by checking whether a randomly chosen pool edge (if such an edge exists) incident on the visited vertex reconnects the tree of the giant component and whether this reconnection is a "good" one (i.e., it does not increase the diameter of the tree). If both checks are successful, then the phase is considered successful and also in turn the NeighborhoodSearch. If all phases fail, then NeighborhoodSearch finishes unsuccessfully, the activation epoch ends, and a total reconstruction is executed. If more than one of the phases are successful, then the replacement edge which is closer to the root of the tree is selected.

A *retirement epoch* starts when the previous activation epoch ends, or when a total reconstruction fails. During the execution of a retirement epoch, the algorithm simply calls another dynamic connectivity algorithm to perform the operations. A retirement epoch lasts for (at least) $cn\log^2 n$ operations ($c > 1$), after which a total reconstruction is performed. If the total reconstruction is successful, then a new activation epoch is started; otherwise, the retirement epoch continues for another $cn\log^2 n$ operations. The process is repeated until a successful total reconstruction occurs.

In the implementation of the NRSY algorithm, henceforth `NRSY`, the `HK` implementation of [11.3] was used in the retirement epochs. In the activation

epoch, each vertex maintains both a set of pool edges and a priority queue of retired edges incident to it. A counter measuring the number of operations in each epoch is maintained. When an edge becomes retired, its priority takes the value of this counter. Reactivation of retired edges occurs only before the deletion of a tree edge, i.e., before the execution of the NEIGHBORHOOD-SEARCH, which is the only procedure for which it is important that several pool edges must exist. Reactivation is performed if the difference of the priority of a retired edge and the current value of the operations counter is larger than $\log n$.

The experiments conducted in [11.22] concentrated on random inputs, since the main goal was to investigate the average-case performance of the NRSY algorithm which guarantees good performance only for these kinds of inputs. The random inputs considered were similar to those generated in [11.3] (cf. Section 11.2.1.2.a). The experiments showed that for long sequences of operations, `NRSY` is better than `HK` and `fast-query`, except for the case where the initial number of edges is small (i.e., $m < n$). For medium sequences of operations, `NRSY` and `HK` perform similarly when m is close to n, but `NRSY` is better as the graph becomes denser. Finally, for short sequences, `HK` and `fast-query` outperform `NRSY`, except for the case of non-sparse graphs. The above behaviour is due to the fact that `NRSY` spends most of its time on activation epochs as the graph becomes denser, while in sparser graphs `NRSY` alternates between retirement epochs and total reconstructions; the overhead imposed by the latter makes `NRSY` slower than `HK` in such cases. In conclusion, `NRSY` is the fastest implementation for random sequences of updates on non-sparse random graphs.

The source code of the above implementations is available from `http://www.ceid.upatras.gr/~faturu/projects.htm`.

11.2.1.2.c *The Implementation by Iyer et al.* [11.44]

The main goal of that paper was to investigate in practice the logarithmic improvement of the fully dynamic algorithm by Holm et al. [11.40], henceforth HDT, over the HK algorithm. To this end, the experimental study built upon the one by Alberts et al. [11.3] and resulted in enhancing the data sets for dynamic connectivity in several ways.

The HDT algorithm maintains a spanning forest F of the current graph G. HDT has many similarities with the HK algorithm, but it differs in the way deletions of tree edges are handled. More precisely, the difference lies in how the levels are organized and how the edges are moved between levels. Both algorithms search for replacement edges at levels no higher than that of the deleted edge. Lower levels contain more important edges (and sparser parts of the graph), while less important edges are moved to higher levels (which contain denser parts of the graph). However, the HDT algorithm starts by considering edges at the bottom level and pushes them up as they are found to be in dense components, while the HK algorithm allows edges to float up automatically but pushes them to lower levels as they are found to be in

sparse components. Since the HDT algorithm also uses ET-trees to maintain the trees in the forest F, it turns out that queries, edge insertions, and non-tree edge deletions are done in a manner similar to that in the HK algorithm. Before giving the details of the deletion of a tree edge, we have to explain how the levels are organized.

Similarly to the HK algorithm, the HDT algorithm assigns to each edge e a level $l(e) \leq L = \log n$, and let F_i denote the subforest of F induced by the edges with level at least i ($F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$). Two invariants are maintained:

(i) F is a maximum w.r.t. l spanning forest, i.e., if (x, y) is a non-tree edge, then x and y are connected in $F_{l(x,y)}$.
(ii) The maximum number of nodes in a tree of F_i is $\lfloor n/2^i \rfloor$.

Initially, all edges have level 0 and as the algorithm proceeds their level is increased (but never decreased). The level of a non-tree edge is increased when it is discovered that its endpoints are close enough in F to fit in a smaller tree on a higher level. The increment of the level of a tree edge should be done with care as it may violate the second invariant. As in the HK algorithm, when a tree edge $e = (x, y)$ with level $l(e) = i$ belonging to a tree T is deleted, we have to find a replacement edge, i.e., an edge which rejoins the two subtrees T_1 and T_2 resulted from the deletion of e. Let T_1 be the smaller subtree. Since $|T| \leq \lfloor n/2^i \rfloor$, it follows that $|T_1| \leq \lfloor n/2^{i+1} \rfloor$. Hence, all edges of T_1 can increase their level to $i + 1$ preserving the invariants. All non-tree edges of T_1 with level i are visited until either a replacement edge is found (in which case we stop), or all edges have been considered. In the latter case, the level of the non-tree edge is increased to $i + 1$ (both its endpoints belong to T_1). If all non-tree edges have been visited without finding a replacement edge, then the procedure is applied recursively on level $i - 1$.

Heuristics are also considered for HDT but, contrary to [11.3], they are restricted only to those which do not invalidate the worst-case time bounds. Following the HK algorithm, the first heuristic considered is sampling; i.e., before edges are promoted to a higher level, a number of incident non-tree edges is randomly sampled and tested for a replacement edge. The second heuristic is similar to the simplified version of the HK algorithm in [11.3]: truncate levels. This is accomplished by exhaustively traversing the (small) trees at higher levels when searching for a replacement edge which gives rise to fewer levels.

The main goal in the design of the experimental setup in [11.44] was to exhibit in practice the asymptotic $O(\log n)$ improvement of the HDT algorithm over the HK one. This could be achieved by designing inputs for which the HK algorithm would indeed match its worst-case time. Consequently, the experimental test set in [11.44] considers three different types of inputs, one random and two structured ones.

The random inputs are similar to those considered in [11.3] (cf. Section 11.2.1.2.a). Since both the HK and HDT algorithms can delete non-tree edges

rather easily, the conducted experiments concentrated on values of m that are close to n, i.e., $m \in \{n/2, 2n\}$, and are among the most interesting regarding the structure of random graphs [11.9].

The structured inputs are split into two groups: two-level inputs (two-level random graphs and two-level semi-random graphs), and worst-case inputs .

The *two-level inputs* are similar to the so-called "non-random inputs" in [11.3]. As mentioned above, the "non-random input" of [11.3] is roughly a path of cliques where only path (i.e., inter-clique) edges are inserted and deleted. A *two-level random graph* is a sparse random graph of cliques. More precisely, k cliques, each of c vertices, are generated ($n = kc$) and are interconnected by $2k$ randomly chosen inter-clique edges. The operation sequence consists of random insertions and deletions of inter-clique edges. This class of inputs is interesting not only as a difficult case for the algorithms (as tree edges are constantly inserted and deleted), but also as an input that exhibits the clustering behaviour which motivated the development of both algorithms. Moreover, it reflects a kind of hierarchical structure that is encountered in several physical networks (e.g., road networks consisting of highways and city streets, computer networks consisting of local area networks interconnected by wide area backbones, etc). *Semi-random graphs* are random graph instances which are strongly correlated over time. Initially a fixed number ($n/2$ or $2n$) of candidate edges is chosen and then random insertions and deletions are performed from this set only. This class is perhaps more interesting than pure random graphs when modeling network applications where links fail and recover, since usually the network is fixed and it is the fixed edges which vanish and return. By replacing each vertex of a semi-random graph with a clique, we can create a *two-level semi-random graph*.

Worst-case inputs aimed at forcing the algorithms to tighten their worst-case time bounds. While it appears difficult to construct a worst-case input for the HK algorithm, Iyer et al. [11.44] succeeded to construct one for the HDT algorithm. Such an input causes HDT to promote $O(n)$ edges through all levels for $O(\log n)$ times during a sequence of $O(n)$ operations.

The worst-case input for HDT is a 4-ary tree with leaf siblings connected to each other and an update sequence constructed as follows. Let $S(k)$ be an operation sequence at the end of which all edges below (and including) tree-level k are at a level greater or equal to k in the HDT data structure. $S(k)$ is defined recursively as follows: (i) Run $S(k-1)$. (ii) Each vertex x at tree-level $k-1$ selects two of its child edges, deletes and re-inserts them. This causes the promotion of a deleted edge (x, y) and of all edges in the subtree rooted at y. (iii) Run again $S(k-1)$ to bring back to level $k-1$ the two child edges which were deleted and re-inserted. (iv) Run step (ii) with the other two child edges.

It is not difficult to see that the sequence $S(\log_4 n)$ will cause $\Theta(n)$ tree edges at tree-level $\log_4 n$ to be promoted through $\Theta(\log n)$ levels resulting in a total time bound of $O(\log^2 n)$.

The implementations of HK and HDT algorithms in [11.44] are tuned by two parameters s and b, and hence are referred to as $\{$`HK`,`HDT`$\}(s, b)$: s denotes the sample size and b denotes that there are only $\log n - \log b$ levels. Hence, `HK`$(16 \log^2 n, 0)$ denotes the `HK` implementation of [11.3], `HK`$(20, n)$ denotes the `HK-var` implementation in [11.3], while `HDT`$(0, 0)$ denotes the implementation of the original HDT algorithm. The implementations of HK and HDT algorithms were also compared to the `fast-update` and `fast-query` implementations developed in [11.3].

The main conclusion of the experimental study in [11.44] is that the heuristics proved to be rather beneficial and that the HDT algorithm with heuristics dominates the HK algorithm. This is due to the repeated rebuildings performed by the latter and the fact that in two-level inputs the HDT algorithm searches through a clique less often than HK. More precisely, for random inputs, where the initial graph has $m \in \{n/2, 2n\}$ edges and random sequences of insertions and deletions are performed such that the graph has always no more than m edges, `HDT`$(0, n)$ (i.e., just an ET-tree) achieves the best performance, followed closely by `HK`$(20, n)$; this is basically due to the truncation of levels. In the case of two-level inputs, `HDT`$(256, 0)$ – when $k < c$ – and `HDT`$(256, 64)$ – when $k \geq c$, or when a semi-random graph is considered – are the fastest. In the former case ($k < c$) this is because most of the inter-clique edges are at level 0 and hence sampling for a replacement most probably will succeed, while in the latter case this is due to the fact that the HDT algorithm searches through a clique less often than HK and due to the overhead introduced by the regular rebuildings performed by HK. Finally, in the worst-case input for HDT, the `HDT`$(256, 64)$ variant achieves the best performance.

The source code of the above implementations is available from `http://theory.lcs.mit.edu/~rajiyer/papers/IKRTcode.tar.gz`.

11.2.1.3 Lessons Learned. The above experimental studies allowed us to gain a deeper insight regarding existing dynamic connectivity algorithms and their practical assessment. In particular:

- The experiments in [11.3] fully confirmed the practicality of sparsification as well as its average-case analysis, thus providing valuable knowledge for the suitability of the theoretical model used and its limitations.
- The heuristic improvements of the HK and HDT algorithms, regardless of whether they do respect the asymptotic time bounds [11.44] or not [11.3], proved to be very useful in practice.
- The study in [11.22] showed that complicated algorithms can sometimes be useful in practice, while the study in [11.44] showed that a logarithmic improvement in asymptotic performance can still have a practical impact.

– The data sets developed in [11.3] formed the cornerstone upon which the subsequent experimental studies were based. We feel that these data sets along with their considerable expansion and elaboration in [11.44] yield an important benchmark for the testing of other dynamic algorithms.

11.2.2 Dynamic Minimum Spanning Tree

11.2.2.1 Theoretical Background — Problem and History of Results. The minimum spanning tree (MST) of a graph $G = (V, E)$, whose edges are associated with real-valued weights, is a spanning tree of minimum total weight. In the dynamic minimum spanning tree problem, we would like to maintain the MST in a graph G that undergoes a sequence of updates (edge insertions and edge deletions).

According to Frederickson [11.25], the first results for fully dynamic MST are attributed to Harel (1983) and achieve an update time of $O(n \log n)$. The first breakthroughs were given by Frederickson in [11.25, 11.26]; in those papers fully dynamic MST algorithms were presented with a running time per update ranging from $O(m^{2/3})$ to $O(m^{1/2})$. As explained in Section 11.2.1.2.a, sparsification by Eppstein et al. [11.19] reduces these running times to be in the range from $O(n^{2/3})$ to $O(n^{1/2})$. A further improvement was achieved by Henzinger & King [11.38] who gave a fully dynamic algorithm with $O(n^{1/3} \log n)$ amortized update bound. Finally, the first polylogarithmic update bound was given by Holm et al. [11.40]; they presented a fully dynamic MST algorithm with amortized update time $O(\log^4 n)$.

11.2.2.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic MST algorithms. In chronological order, these are the works by Amato et al. [11.4] and by Cattaneo et al. [11.10].

The former paper is a follow up of the study in [11.3] and investigates the practical performance of both Frederickson's algorithms [11.25, 11.26] and of stable sparsification on top of dynamic algorithms [11.19]. The latter paper enhances the study in [11.4] by bringing the algorithm by Holm et al. [11.40] into play and aims at investigating its practicality in comparison to the implementations in [11.4] as well as to new simple algorithms based on static approaches.

Throughout the rest of this section, let $G = (V, E)$ be the input graph and let T be its minimum spanning tree.

11.2.2.2.a *The Implementation by Amato et al.* [11.4]

The main goal of the first experimental study for dynamic MST algorithms was:

1. To investigate the practicality of stable sparsification, i.e., spasification on top of dynamic algorithms [11.19].

2. To compare sparsification-based algorithms with Frederickson's algorithms [11.25, 11.26] for dynamic MST.
3. To investigate the practicality of the dynamic algorithms in comparison to simple-minded algorithms based on static approaches that were easy to implement and likely to be fast in practice.

The algorithms by Frederickson are based on appropriately grouping the vertices of T into *vertex clusters* (set of vertices which induce a connected subgraph of T) such that suitable partitions on V are defined either directly (yielding *balanced* or *restricted partitions*) or indirectly by recursive applications of clustering (yielding *topology trees* and *2-dimensional topology trees*). These partitions allow for an encoding of the MST which can be efficiently updated after a dynamic change in G.

Two different partitions are given in [11.25, 11.26]. The first partition [11.25] is called a *balanced partition of order* z and is simply a partition of V into vertex clusters of cardinality between z and $3z-2$. The second partition [11.26] is called *restricted partition of order* z as it sets more requirements on how clustering is done: (i) each set in the partition yields a vertex cluster of external degree[2] at most 3; (ii) each cluster of external degree 3 has cardinality 1; (iii) each cluster of external degree less than 3 has cardinality at most z; and (iv) no two adjacent clusters can be combined and still satisfy the above. Both partitions have $O(m/z)$ clusters. The maintenance of each partition during edge insertions and deletions allows it to dynamically maintain the MST of a graph in time $O(z+(m/z)^2) = O(m^{2/3})$ [11.25]. This method yields the first two algorithms implemented in [11.4], namely `FredI-85` (based on the balanced partition) and `FredI-91` (based on the restricted partition).

Although the asymptotic behaviour of both partitions is identical, the experiments conducted in [11.4] revealed several differences between balanced and restricted partitions w.r.t. their practical behaviour that favor the former: (a) the number of clusters generated by a balanced partition is much less than those generated by a restricted partition; (b) there is a smaller number of splits and merges of clusters (which are expensive operations) in a balanced partition and hence clusters have a longer lifetime; and (c) the average number of affected clusters by an update is substantially smaller in balanced partitions than in restricted partitions. As a consequence, `FredI-85` was always faster than `FredI-91`.

The above observed differences motivated a theoretical and experimental tuning of the parameters of the two partitions resulting in a third partition called *light partition* [11.4]. This partition is a relaxed version of the restricted partition, namely: (i) each cluster is of cardinality at most z; and (ii) no two adjacent clusters can be combined and still satisfy the above. Since a light partition is a relaxation of the restricted partition, its number of clusters (at least initially) cannot be more than those of the restricted partition,

[2] Number of edges having their other endpoint at a different cluster.

i.e., $O(m/z)$. The problem, however, with light partition is that there is no guarantee that this number is preserved throughout any sequence of edge insertions and deletions. Consequently, the worst-case update bound of a dynamic MST algorithm based on light partitions are worse than $O(m^{2/3})$. On the other hand, the experiments in [11.4] showed that in practice the number of clusters in a light partition does not increase much beyond its initial number of $O(m/z)$ as the number of edge updates increases. To further increase its efficiency, the light partition was further "engineered" in [11.4] by introducing a lazy update scheme for the expensive update of the light partition: the partition is updated only in the case of tree edge deletions (i.e., in the case where there is certainly a change in the current MST). Edge insertions are handled in $O(\log n)$ time by recomputing the MST of the current graph using the dynamic tree data structure of Sleator & Tarjan [11.61]. Hence, a cluster that would otherwise be affected by several edge insertions and deletions, is now updated only at the time of a tree edge deletion. This lazy update scheme along with a light partition of order $\lceil m^{2/3} \rceil$ yields another implementation for the dynamic MST problem referred to as `FredI-Mod` [11.4]. The experiments in [11.4] showed that `FredI-Mod` was always significantly faster than both `FredI-85` and `FredI-91`.

The recursive application of balanced or restricted partitions yields different types of partitions that end up in the so-called topology tree. A *multi-level balanced partition* has the following properties: (i) the clusters at level 0 contain a single vertex; (ii) a cluster at level $i \geq 1$ is either a cluster at level $(i-1)$ of external degree 3 or the union of at most 4 clusters (according to some rules) at level $(i-1)$; (iii) there is exactly one cluster at the topmost level. A *multi-level restricted partition* obeys the same properties, except for (ii) which is stated as: the clusters at level $i \geq 1$ form a restricted partition of order 2 w.r.t. the tree obtained after shrinking all clusters at level $(i-1)$. Note that this rule makes a multi-level restricted partition to be defined in a much simpler way than a multi-level balanced partition. A *topology tree* is a tree which represents the above multi-level partitions, i.e., a *balanced* (resp. *restricted*) *topology tree* is a tree for which a node at level i represents a cluster at level i of a balanced (resp. restricted) multi-level partition, and the children of a node at level $i \geq 1$ are the clusters at level $(i-1)$ whose union gives the cluster at level i. It is easy to see that both topology trees have height $O(\log N)$, where N is the total number of nodes in the tree. In [11.25, 11.26] it is shown that updating any topology tree after an edge insertion, edge deletion, or edge swap, takes $O(\log N)$ time.

By using a topology tree to represent each cluster in a (balanced or restricted) partition of order z, we can get an $O(z + (m/z)\log(m/z)) = O(\sqrt{m \log m})$ time algorithm for the dynamic MST problem. This yields two implementations in [11.4], namely `FredII-85` (balanced partitions and topology trees) and `FredII-91` (restricted partitions and topology trees). The experiments conducted in [11.4] showed that `FredII-91` was slightly faster than

`FredII-85`, mainly due to the simpler clustering rules which the restricted topology trees and multi-level partitions obey. In [11.4], a hybrid solution was also investigated involving a suitable combination of a balanced partition of order z with restricted topology trees. However, even this hybrid solution turned out to be slower than `FredI-85` for most inputs they considered.

To efficiently maintain information about the non-tree edges, Frederickson introduces the *2-dimensional topology trees* which are defined by pairs of nodes in a topology tree. For every pair of nodes V_a and V_b at the same level of a topology tree, there is a node labeled $V_a \times V_b$ in the 2-dimensional topology tree which represents the non-tree edges of G having one endpoint in V_a and the other in V_b. If V_a (resp. V_b) has children V_{a_j}, $1 \leq j \leq p$ (resp. V_{b_k}, $1 \leq k \leq q$) in the topology tree, then $V_a \times V_b$ has children the nodes $V_{a_j} \times V_{b_k}$ ($1 \leq j \leq p$, $1 \leq k \leq q$) in the 2-dimensional topology tree. The use of a 2-dimensional topology tree yields an $O(z + m/z) = O(\sqrt{m})$ time algorithm for the dynamic MST problem. This theoretical improvement, however, does not show up in practice: all implementations that employed 2-dimensional topology trees in [11.4] were much slower than `FredI-85`.

The above implementations were enhanced by applying stable sparsification (i.e., simple sparsification where a fully dynamic algorithm is used to recompute certificates; see Section 11.2.1.2.a) on top of them. More precisely, sparsification was applied on top of `FredI-85` yielding algorithm `Spars(I-85)` with an $O(n^{2/3} \log(m/n))$ worst-case time bound, and on top of `FredI-Mod` yielding algorithm `Spars(I-Mod)`.

Finally, a simple fully dynamic algorithm was implemented in [11.4], called `adhoc`, which is a combination of a partially dynamic data structure – based on the dynamic trees of Sleator & Tarjan [11.61] – and LEDA's static MST algorithm called `Min_Spanning_Tree`, which is a fine-tuned variant of Kruskal's algorithm based on randomization with an average-case time of $O(m + n \log^2 n)$ and a worst-case time of $O(m \log m)$. The `adhoc` algorithm maintains two data structures: the MST T of G as a dynamic tree, and a priority queue Q which stores all edges of G according to their weight. When a new edge is inserted, `adhoc` updates T and Q in $O(\log n)$ time. When an edge is deleted from G, it is first deleted from Q in $O(\log n)$ time. If it was a non-tree edge, nothing else happens. Otherwise, `adhoc` calls `Min_Spanning_Tree` on the edges of Q. Consequently, `adhoc` requires $O(\log n)$ time plus the time of `Min_Spanning_Tree` in the case where a tree edge is deleted. If the edge to be deleted is chosen uniformly at random from G, then this expensive step occurs with probability n/m resulting in an average running time of $O(\log n + (n/m)(m + n \log^2 n)) = O(n + (n \log n)^2/m)$ for `adhoc`. Hence, it is natural to expect that its running time decreases as the graph density increases, a fact that was confirmed by the experiments in [11.4].

As mentioned above, in all experiments which carried out in [11.4], `FredI-85` was consistently the fastest among the six algorithms derived by the approaches in [11.25, 11.26]. Hence, this implementation was cho-

sen to be compared experimentally with the rest of implementations, namely `FredI-Mod`, `Spars(I-85)`, `Spars(I-Mod)`, and `adhoc`.

Experiments were run on both random and non-random inputs. Random inputs aimed at investigating the average-case performance of the implemented algorithms and confirming the average-case complexity of `adhoc`. Non-random inputs aimed at producing test sets which would make the algorithms to exhibit their worst-case performance.

Random inputs were similar to those considered in [11.3, 11.44] and consisted of random graphs (with random edge weights) and random operation sequences in which edge insertions were uniformly mixed with edge deletions. Also, the edge weights in each update sequence were chosen at random. Non-random inputs were generated by first constructing a graph, then computing its MST, and finally deleting the edges of the MST one at a time. Recall that a tree edge deletion is the most expensive operation in any of the dynamic MST algorithms considered.

In all experiments with random inputs, `adhoc` was almost always the fastest algorithm; only in the case of (initially) sparse graphs on a large number of vertices, `FredI-Mod` was faster. Among the dynamic algorithms, `FredI-Mod` was consistently faster than any of the other implementations followed by `FredI-85`. The implementations based on sparsification were the slowest. In non-random inputs, however, a "reverse" situation was reported. That is, `Spars(I-Mod)` was the fastest algorithm, followed by `FredI-Mod`, and `adhoc` was by far the worst. This was more or less expected, since random edge deletions (especially in non-sparse graphs) will most probably be non-tree edge deletions which makes `adhoc` superior for random inputs; on the contrary, tree edge deletions make `adhoc` exhibit its worst-case performance since they cause it to recompute the MST.

The different behaviour of the implementations based on sparsification can be explained as follows. Sparsification spreads the information about a graph G into smaller sparse subgraphs. Even if a random update may not change the MST of the entire G, it may happen that some of the smaller subgraphs have to change their MST and this shows up in random inputs. On the other hand, a bad update operation (tree edge deletion) will not make a big difference w.r.t. a good update, since any update spreads on a logarithmic number of smaller subgraphs. Consequently, this spreading is simultaneously an advantage (for non-random inputs) and a disadvantage (for random inputs) of sparsification. A final conclusion from the experimental study of [11.4] was the efficiency of `FredI-Mod` which was consistently faster than `FredI-85` and `Spars(I-85)` even for non-random inputs, mainly due to the more relaxed nature of light partitions.

The source code of the above implementations is available via anonymous ftp from `ftp.dia.unisa.it` in the directory `pub/italiano/mst`.

11.2.2.2.b *The Implementation by Cattaneo et al.* [11.10]

The main goal of that study was to investigate the practicality of the recent fully dynamic MST algorithm by Holm et al. [11.40], henceforth HDT$_{mst}$, and comparing it with Frederickson's dynamic algorithms, sparsification-based algorithms and new simple algorithms. To this end, the experimental study in [11.10] built upon the studies by Amato et al. [11.4] and by Iyer et al. [11.44], and resulted in considerably enhancing the quiver of dynamic MST implementations as well as our knowledge on their practicality.

The HDT$_{mst}$ algorithm is based on a decremental algorithm for MST which has many similarities with the HDT algorithm for dynamic connectivity (cf. Section 11.2.1.2.c). Then, using a rather general construction which was first introduced in [11.38], the decremental algorithm is converted into a fully dynamic one.

The decremental MST algorithm can be obtained by the HDT algorithm by doing two very simple changes. First, a minimum spanning forest (instead of any spanning forest) F is maintained. Second, there is a different way with which non-tree edges are considered for replacement edges, when a tree edge is deleted. Instead of an arbitrary order, the non-tree edges incident to the smaller subtree are considered in order of non-decreasing weight.

The partition of edges into levels is done similarly to that in the HDT algorithm (cf. Section 11.2.1.2.c). In addition to the two invariants of the HDT algorithm, the following invariant is maintained.

(iii) If e is the heaviest edge on a cycle C, then e has the lowest level on C.

It is not hard to see that the above invariant ensures that the minimum weight edge among all replacement edges is the lightest edge among all replacement edges on the highest possible level.

The approach for converting the above decremental algorithm to a fully dynamic one is as follows. A set of data structures $A = A_1, A_2, \ldots, A_s$ is maintained, where $s = \lceil \log n \rceil$ and each A_i is a subgraph of G. Let F_i be the local spanning forest maintained in A_i. Edges in F_i are referred to as local tree edges while those in F are referred to as global tree edges. All edges of G are in at least one A_i, hence $F \subseteq \bigcup_i F_i$. The algorithm maintains the invariant that for each global non-tree edge $e \in G - F$, there is exactly one i such that $e \in A_i - F_i$ and if $e \in F_j$, then $j > i$. The minimum spanning forest F is maintained as a dynamic tree data structure of Sleator & Tarjan [11.61] (ST-tree) and also as an Euler-Tour tree (ET-tree) of Henzinger & King [11.36] (in order to easily find replacement edges).

Before describing the edge insertion and deletion operations, we have to describe an auxiliary procedure, called *Update*(A, D), which updates the data structure A with a set of edges D. Let $B_j = \bigcup_{k \leq j}(A_k - F_k)$, i.e., the set of local non-tree edges in all A_k, for $k \leq j$. The procedure works as follows. It finds the smallest j such that $|(D \cup B_j) - F| \leq 2^j$ and sets $A_j = F \cup D \cup B_j$. Then, it initializes A_j as a decremental MST data structure and sets $A_k = \emptyset$ for all $k < j$.

The insertion of a new edge $e = (x, y)$ is carried out as follows. If x and y are in different trees in F, then simply add e to F. Otherwise, find the heaviest edge f in the x-y path. If e is heavier than f, then call *Update*(A, e). Otherwise, replace f with e in F and call *Update*(A, f).

The deletion of an edge e is done as follows. The edge e is deleted from all A_i which contained e and let R be the set of the collected replacement edges returned by all decremental MST data structures. If e is a global tree edge, then search in R (using the ET-tree representation of F) to find the lightest edge which reconnects F. Finally, call *Update*(A, R).

The crucial point in the update procedure is the initialization of A_j. To achieve this efficiently, Holm et al. [11.40] perform a contraction of some local tree paths (those which are not incident on any non-tree edge) using a special data structure they introduce, called *top trees*. This allows them to bound the initialization work in each update.

The implementation of the HDT$_{mst}$ algorithm in [11.10] is not based on the `HDT` implementation of Iyer et al. [11.44], because Cattaneo et al. [11.10] found the latter specifically targeted and engineered for the dynamic connectivity problem and conversion of this code for dynamic MST appeared to be a difficult task. Hence, they provided a completely new implementation that is better suited for dynamic MST. In their implementation they follow closely the above described HDT$_{mst}$ algorithm with the exception that they do not use top trees for path compression, since top trees were consuming a lot of memory. They initially used ST-trees instead, and later on observed that by completely omitting path compression a considerable gain in performance was obtained [11.23]. The resulting implementation is called `HDT`.

The simple algorithms considered in [11.10] are a kind of fast dynamization of Kruskal's algorithm. One algorithm is based on ST-trees and the other on ET-trees.

The first algorithm maintains the MST T as an ST-tree and the non-tree edges are maintained sorted in a binary search tree NT.

The insertion of a new edge (x, y) is similar to the insertion procedure of the fully dynamic case: if (x, y) should replace an existing tree edge f (which can be easily checked in $O(\log n)$ time using the ST-trees), then f is deleted from T and is inserted into NT. Otherwise, (x, y) does not belong to the MST and is inserted into NT. Clearly, insertion can be accomplished in $O(\log n)$ time.

The deletion of an edge (x, y) depends on whether it is a non-tree or a tree edge. In the former case, (x, y) is simply deleted from NT in $O(\log n)$ time. In the latter case, the deletion of (x, y) disconnects T into T_x (the subtree containing x) and T_y (the subtree containing y) and a replacement edge has to be found. To accomplish this, non-tree edges in NT are scanned in non-decreasing weight order in a manner similar to Kruskal's algorithm: if a non-tree edge (s, t) reconnects T_x and T_y (a fact which can be easily checked using the findroot(s) and findroot(t) operations of ST-trees), then

the scanning is terminated; otherwise, the non-tree edge is discarded. Clearly, the total time of the deletion operation is $O(k \log n)$, where k is the number of scanned non-tree edges, which is $O(m \log n)$ in the worst case.

The implementation of the above algorithm is referred to as `ST`. Since `ST` can be implemented in a few lines of code using fast (and simple) data structures, it is expected that it performs very well in practice, especially when the update sequence contains a few tree edge deletions or when a tree edge deletion results in a small set of possible replacement edges. This fact was confirmed in most of the experiments, except for the case of sparse graphs i.e., with m close to n. In these cases, and when dealing with random graphs, a random edge deletion will most probably be a tree edge that disconnects the MST and consequently will cause the scanning of many non-tree edges until the proper replacement edge is found (if any). Actually, it turned out that most of the time in these cases was spent in executing findroot operations.

Motivated by this difficult case of `ST`, Cattaneo et al. [11.10] designed another variant, called `ET`, which in addition uses ET-trees (as they have been implemented in [11.3] and supported by the randomized search trees of [11.6]). More precisely, the information about tree edges is kept both on an ST-tree and on an ET-tree. ET-trees are used only during the deletion operation to check whether a non-tree edge reconnects the MST. It should be pointed out that ET-trees have the same asymptotic time bounds with ST-trees (cf. Section 11.2.1.2.a). Since findroot operations in randomized search trees are expected to be faster than those in ST-trees, it is consequently expected that `ET` is faster than `ST` on sparse graphs. However, it is also expected that the overhead of maintaining both ET-trees and ST-trees will show up when the findroot operations are not any more the bottleneck. Both expectations were confirmed by the experiments conducted in [11.10].

The above implementations `HDT, ST` and `ET` were experimentally compared with `Spars(I-Mod)` from [11.4] (cf. Section 11.2.2.2.a). The experimental test set was built upon the ones used in [11.4] and [11.44]. In particular, both random and structured inputs were considered.

The random inputs were identical to the random inputs used in [11.4] (cf. Section 11.2.2.2.a). The structured inputs consisted of semi-random inputs, two-level random inputs, and worst-case inputs, all generated in way very similar to that in [11.44] (cf. Section 11.2.1.2.c).

Semi-random inputs consisted of semi-random graphs with edge weights chosen at random and where update operations are chosen from a fixed set of edges in a way identical to that in [11.44].

Two-level random inputs consisted of two-level random graphs (k cliques, each one of c vertices, interconnected by $2k$ randomly chosen inter-clique edges) and edge weights chosen at random. The operation sequence was of two types. The first involved only insertions and deletions of inter-clique edges, as it happens in [11.44]. The second type involved insertions and deletions of

edges inside a clique (intra-clique edges) and mixed updates between intra-clique and inter-clique edges.

Finally, a worst-case input for HDT_{mst} was constructed by suitably adapting the worst-case input of HDT in [11.44] for dynamic connectivity (cf. Section 11.2.1.2.c).

In the experiments with random inputs, `ET` was the fastest implementation in sparse graphs followed by `ST`, even in sequences with a high percentage of tree edge deletions. In non-sparse graphs, `ET` and `ST` had almost identical behaviour, with `ET` being slightly worse due to the additional overhead introduced by maintaining both ST-trees and ET-trees. `HDT` was slower than `Spars(I-Mod)` when the graph is sparse, but it takes over as the edge density increases and the overhead of the sparsification tree in `Spars(I-Mod)` shows up. These outcomes are quite interesting, since they match those of other studies [11.4, 11.44] (cf. Sections 11.2.2.2.a and 11.2.1.2.c, respectively) and show that ET-trees are a particularly valuable data structure for dynamic problems on random graphs.

Similar behaviour was reported for the case of semi-random inputs when the cardinality of the fixed set of edges (from which updates are chosen randomly) is small. However, when the cardinality of the fixed set of edges is increased, `ET` still remains the fastest followed closely by `Spars(I-Mod)`. The other two algorithms are considerably penalized in performance, with `ST` being by far the slowest – apparently due to overhead of findroot operations.

In the case of two-level inputs, both `ET` and `ST` were not competitive, because of the considerable overhead introduced by the deletion of inter-clique edges. The `HDT` implementation was faster, regardless of the clique size, in all cases where there were either no intra-clique edge deletions, or very few of them. However, as the number of intra-clique edge deletions increases, `Spars(I-Mod)` improves over `HDT`.

In the experiments with the worst-case input, `HDT` is penalized by the update sequence in executing unnecessary promotions of edges among levels and is worse than `ET` or `ST` which actually achieve their best performance of $O(\log n)$ (as non-tree edges do not exist to be considered for replacement edges). `Spars(I-Mod)` also suffered a significant performance loss by this input (it was the slowest), because a tree edge deletion carries the burden of the overhead imposed by the implementation of light partition.

11.2.2.3 Lessons Learned. The above experimental studies considerably enhance our knowledge regarding the practicality of dynamic MST algorithms.

The work by Amato et al. [11.4] is an exemplary study of algorithm engineering. Thorough experimentation helped in identifying the best algorithm in practice (an otherwise extremely difficult task) from a collection of identically behaving algorithms w.r.t. the asymptotics. Experimentation also revealed the bottlenecks in performance, leading to the introduction of heuristics followed by careful fine-tuning. This considerably improved the practical

performance of theoretically inferior algorithms. On the other hand, theoretically superior algorithms achieved through the use of several layers of complicated data structures turn out not to be useful in practice.

The work by Cattaneo et al. [11.10] is a follow up which on the one hand provides an implementation of the theoretically best algorithm, and on the other hand helps in identifying the cases in which that algorithm can be of real practical value.

Finally, as in the case of dynamic connectivity, the carefully designed data sets exhibited the differences in the average and worst case performances of the implemented algorithms. To this end, the data sets for dynamic connectivity developed in [11.3, 11.44] turned out to be an excellent starting point.

11.3 Dynamic Algorithms for Directed Graphs

The implementation studies known for dynamic problems in directed graphs (digraphs) concern transitive closure and shortest paths.

11.3.1 Dynamic Transitive Closure

11.3.1.1 Theoretical Background — Problem and History of Results. Given a digraph $G = (V, E)$, the *transitive closure* (or *reachability*) problem consists in finding whether there is a directed path between any two given vertices in G. We say that a vertex v is *reachable* by vertex u iff there is a (directed) path from u to v in G. The digraph $G^* = (V, E^*)$ that has the same vertex set with G but has an edge $(u, v) \in E^*$ iff v is reachable by u in G is called the *transitive closure* of G; we shall denote $|E^*|$ by m^*. If v is reachable from u in G, then we call v a *descendant* of u, and u an *ancestor* of v. In the following we denote by $DESC[v]$ the set of descendants of v.

There are several partially dynamic algorithms for transitive closure, and some recent fully dynamic ones. All algorithms create a data structure that allows update operations (edge insertion/deletion) and query operations. A query takes as input two vertices u and v and can be either `Boolean` (returns "true" if there is a u-v path, otherwise "false") or `Path` (returns in addition the actual path if it exists). In the following, let G_0 be the initial digraph (before performing a sequence of updates) having n vertices and m_0 edges. We start with the partially dynamic algorithms.

The first partially dynamic algorithms were given by Ibaraki and Katoh [11.41]. Their incremental (resp. decremental) algorithm supported any number m of edge insertions (resp. deletions) in $O(n(m + m_0)^*)$ (resp. $O((n + m_0)m_0^*)$) time and answered `Boolean` queries in $O(1)$ time, and `Path` queries in time proportional to the number of edges of the reported path (i.e., both type of queries are answered optimally). These bounds were later improved by Italiano [11.42, 11.43], Yellin [11.66], La Poutré & van Leeuwen [11.49],

and Cicerone et al. [11.12]. The decremental part of these algorithms apply only to directed acyclic graphs (DAGs). Italiano's and Yellin's data structures support both `Path` and `Boolean` queries, while the other two data structures [11.12, 11.49] support only `Boolean` queries. All these partially dynamic algorithms, except for Yellin's, support any number m of edge insertions (resp. deletions) in $O(n(m + m_0))$ (resp. $O(nm_0)$ time); i.e., for $m = \Omega(m_0)$, they achieve an $O(n)$ amortized update bound. `Boolean` and `Path` queries (where applicable) are answered optimally. The algorithm by La Poutré & van Leeuwen [11.49] can be extended to handle edge deletions in general digraphs with an amortized time of $O(m_0)$ per deletion. Also, in [11.32, 11.33] the algorithms by Italiano [11.42, 11.43] were extended so that the decremental part applies to any digraph. While the amortized time per edge deletion remains $O(m_0)$, this new algorithm is able to support `Path` queries optimally. The above data structures by Italiano, La Poutré & van Leeuwen, and Cicerone et al. can be initialized in $O(n^2 + nm_0)$ time and require $O(n^2)$ space.

Yellin's data structure has different initialization bounds depending on whether it supports `Path` queries or not. More precisely, the initialization time and space bounds are the same as those of the other algorithms, if Yellin's data structure supports only `Boolean` queries. If in addition `Path` queries are supported, then both time and space bounds become $O(n^2 + dm_0^*)$, where d is the maximum outdegree of G_0. In either variant of Yellin's algorithm: (i) Queries are supported optimally. (ii) The incremental part requires $O(d(m_0 + m)^*)$ time to process a sequence of m edge insertions starting from an initial digraph G_0 and resulting in a digraph G; d is the maximum outdegree of G. (iii) The decremental version requires $O(dm_0^*)$ time to process any sequence of m edge deletions; d is the maximum outdegree of G_0.

Henzinger & King in [11.37] gave a decremental randomized algorithm which is initialized in $O(n^2 + nm_0)$ time and space, supports any sequence of edge deletions in $O(m_0 n \log^2 n)$ expected time, and answers `Boolean` queries in $O(n/\log n)$ worst-case time. `Path` queries are supported in an additional time which is proportional to the number of edges of the reported path. Recently, two incremental algorithms were given by Abdeddaim [11.1, 11.2] who showed that if the graph is covered by a set of k vertex-disjoint paths, then any sequence of m edge insertions can be accomplished either in $O(k^2(m_0 + m) + (m_0 + m)^*)$ time [11.1], or in $O(k(m_0 + m)^*)$ time [11.2]. Both algorithms use $O(kn)$ space.

We now turn to the fully dynamic algorithms. The first two fully dynamic algorithms for transitive closure were presented in [11.37]. They are randomized (with one-side error) and are based on the decremental algorithm presented in the same paper. The first (resp. second) fully dynamic algorithm supports update operations in amortized expected time $O(\hat{m}\sqrt{n}\log^2 n + n)$ (resp. $O(n\hat{m}^{0.58}\log^2 n)$), where $\hat{m}$ is the average number of edges in G during the whole sequence of updates. Since $\hat{m}$ can be as high as $O(n^2)$, the update

bounds of [11.37] can be $O(n^{2.5} \log^2 n)$ (resp. $O(n^{2.16} \log^2 n)$). Queries are answered within the same time bounds as those of the decremental algorithm. Khanna, Motwani, and Wilson [11.46] presented a fully dynamic algorithm that achieves a deterministic amortized update bound of $O(n^{2.18})$, when a lookahead of size $\Theta(n^{0.18})$ in the update sequence is allowed. The next fully dynamic algorithm was given by King and Sagert [11.48]. It is a randomized one (with one-side error) supporting updates in $O(n^{2.26})$ amortized time for general digraphs and in $O(n^2)$ worst-case time for DAGs, and supports `Boolean` queries in $O(1)$ time. These bounds were further improved by King in [11.47], who gave a deterministic algorithm with $O(n^2 \log n)$ amortized update time and $O(1)$ `Boolean` query time. Very recently, Demetrescu and Italiano [11.13] outperformed the above approaches by presenting two fully dynamic algorithms: the first is a deterministic one supporting updates in $O(n^2)$ amortized time and `Boolean` queries in $O(1)$ time; the second is a randomized algorithm (with one-side error), it applies only to DAGs, and achieves a trade-off between query and update time. The currently best worst-case time for an update is $O(n^{1.575})$, which achieves a worst-case `Boolean` query time of $O(n^{0.575})$.

11.3.1.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic algorithms for transitive closure. In chronological order, these are the works by Frigioni et al. [11.32, 11.33] and by Abdeddaim [11.2].

The former paper investigates the practicality of several dynamic algorithms (most of them with identical theoretical performance) in partially and fully dynamic settings. The latter paper investigates incremental dynamic algorithms under a specific application scenario (sequence alignment).

11.3.1.2.a *The Implementation by Frigioni et al.* [11.32, 11.33]

The main goal of the first experimental study for dynamic transitive closure was threefold:

1. To investigate the practicality of dynamic algorithms over their static counterparts – especially to very simple and easily implementable ones.
2. To investigate the differences in the practical performance of several dynamic algorithms which appear to have similar asymptotic behaviour.
3. To compare partially dynamic algorithms against fully dynamic ones.

Several partially dynamic and one fully dynamic algorithm were investigated in [11.32, 11.33]. More precisely, the partially dynamic algorithms considered were those by Italiano [11.42, 11.43], Yellin [11.66], Cicerone et al. [11.12], and Henzinger & King [11.37]. The fully dynamic algorithm was one of the two presented in [11.37]. Also, as a consequence of fine-tuning of some of these algorithms, several variants were obtained (including hybridizations of partially dynamic algorithms to yield fully dynamic ones)

which turned out to be quite fast in practice. All these algorithms were compared with other simple-minded approaches that were easy to implement and hence likely to be fast in practice.

The starting point was the implementation of Italiano's algorithms [11.42, 11.43]. The main idea of the data structure proposed in those papers is to associate (and maintain) with every vertex $u \in V$ a set $DESC[u]$ containing all descendants of u in G. Each such set is organized as a spanning tree rooted at u. In addition, an $n \times n$ matrix of pointers, called $INDEX$, is maintained which allows fast access to vertices in these trees. More precisely, $INDEX[i, j]$ points to vertex j in $DESC[i]$, if $j \in DESC[i]$, and it is $Null$ otherwise.

A `Boolean` query for vertices i and j is carried out in $O(1)$ time, by simply checking $INDEX[i, j]$. A `Path` query for i and j is carried out in $O(\ell)$ time, where ℓ is the number of edges of the reported path, by making a bottom-up traversal from j to i in $DESC[i]$.

The insertion of an edge (i, j) is done as follows. The data structure is updated only if there is no i-j path in G. The insertion of edge (i, j) may create new paths from any ancestor u of i to any descendant of j only if there was no previous u-j path in G. In such a case the tree $DESC[u]$ is updated using the information in $DESC[j]$ (deleting duplicate vertices) and accordingly the u-th row of $INDEX$.

The deletion of an edge (i, j) on a DAG G is done as follows. If (i, j) does not belong to any $DESC$ tree, then the data structure is not updated. Otherwise, (i, j) should be deleted from all $DESC$ trees to which it belongs. Assume that (i, j) belongs to $DESC[u]$. The deletion of (i, j) from $DESC[u]$ splits it into two subtrees, and a new tree should be reconstructed. This is accomplished as follows. Check whether there exists a u-j path that avoids (i, j); this is done by checking if there is an edge (v, j) in G such that the u-v path in $DESC[u]$ avoids (i, j). If such an edge exists, then swap (i, j) with (v, j) in $DESC[u]$, and join the two subtrees using (v, j). In such a case, (v, j) is called a *valid replacement* for (i, j), and v is called a *hook* for j. If such an edge does not exist, then there is no u-v path in G and consequently j cannot be a descendant of u anymore: delete j from $DESC[u]$ and proceed recursively by deleting the outgoing edges of j in $DESC[u]$. To quickly find valid replacement edges, an $n \times n$ matrix $HOOK$ is used in the implementation. The entry $HOOK[u, j]$ stores the pointer to the first unscanned item in j's list of incoming edges $IN[j]$, if such an item exists; otherwise, $HOOK[u, j] = Null$. It is also easy to verify that if some $x \in IN[j]$ has already been considered as a tentative hook for j, then it will never be a hook for j in any subsequent edge deletion. The implementation of the above described algorithm is referred to as `Ital`. In [11.32, 11.33], a second version of the above algorithm was implemented by removing recursion in the edge insertion and edge deletion procedures. This yielded implementation `Ital-NR`. The experiments showed that `Ital` and `Ital-NR` have almost identical performances with `Ital` quite often being slightly faster.

Italiano's algorithms were further optimized in [11.33] by providing a fine-tuned third implementation, called `Ital-Opt`, which maintains descendant trees and hook information implicitly. This implementation was consistently faster than `Ital` and `Ital-NR` in all experiments performed.

The above implementations of Italiano's algorithm were adapted to work in fully dynamic environments, although their asymptotic bounds do not hold in this case. The goal was to experimentally compare them with fully dynamic algorithms as it may happen that they work well in practice, a fact that was indeed confirmed by the experiments in [11.32, 11.33]. To handle mixed sequences of updates, Italiano's algorithm has to be modified, since now the insertion of new edges may provide new hook information for some vertices. Consequently, the $HOOK$ matrix has to be reset after each edge insertion that is followed by a sequence of edge deletions. Resetting the $HOOK$ matrix takes $O(n^2)$ time. The reset operation has been incorporated in `Ital` and `Ital-NR` when they are used in a fully dynamic environment. Since the overhead introduced by each reset may be significant (as it was also verified in practice), a *lazy approach* was adopted in [11.32, 11.33]: delay the resetting of an entry of the $HOOK$ matrix until it is required by the algorithm. This lazy approach was incorporated in `Ital-Opt`. Experiments showed a significant improvement upon Italiano's original algorithms (`Ital` and `Ital-NR`) on mixed sequences of updates.

The above ideas were extended in [11.32, 11.33] to develop a new algorithm whose decremental part applies to any digraph, and not only to DAGs. This algorithm (and its implementation) is referred to as `Ital-Gen`.

The algorithm is based on the fact that if we shrink every strongly connected component of a digraph $G = (V, E)$ to a single vertex, called *super-vertex*, then the resulting graph $G' = (V', E')$ is a DAG. The idea is to use Italiano's algorithm to maintain the transitive closure in G' and additional information regarding the strongly connected components (SCCs) which is crucial for the decremental part of the algorithm.

The data structure consists of: (a) a collection of implicitly represented descendant trees (as in `Ital-Opt`); (b) an $n \times n$ Boolean matrix $INDEX$ (as in `Ital`); (c) an array SCC of length n, where $SCC[i]$ points to the SCC containing vertex i; and (d) the SCCs of G as graphs. Furthermore, with each k-vertex SCC two additional data structures are maintained: an array $HOOK$ of length n, where $HOOK[i]$ points to the incoming edge of the SCC used in the (implicitly represented) descendant tree rooted at i; and a *sparse certificate* of the SCC consisting of k vertices and $2k - 2$ edges.

The initialization involves computation of the above data structures where the computation of the SCC and their sparse representatives is performed only for the decremental part of the algorithm, i.e., before any sequence of edge deletions. (If there is no such sequence, then every vertex is taken as a SCC by itself.)

`Boolean` and `Path` queries can be answered in the same time bounds with those of `Ital`, by first looking at the *INDEX* matrix to check whether there exists a path; if yes, then the path can be found using the *HOOK* arrays (which provide the path in G') and the sparse certificate of each SCC (which gives the parts of the required path represented by supervertices in G').

The insertion of an edge (i, j) is done similarly to the original algorithm by Italiano. Deleting an edge (i, j) is done as follows. If (i, j) does not belong to any SCC, then use Italiano's decremental algorithm to delete (i, j) from G'. Otherwise, check if (i, j) belongs to the sparse certificate of the SCC or not. In the latter case, simply remove the edge from the SCC. In the former case, check if the deletion of (i, j) breaks the SCC. If the SCC does not break, we may need to recompute the sparse certificate. If the SCC breaks, then compute the new SCCs, update the implicit data structures and the *HOOK* arrays properly so that the information concerning descendant trees and hooks in the new G' is preserved, and finally apply Italiano's decremental algorithm to delete (i, j) from the new G'. The maintenance of the transitive closure in G' is done by a suitable modification of `Ital-Opt` which facilitates the splitting of SCCs.

To use `Ital-Gen` in a fully dynamic environment, further modifications and optimizations need to be made. Instead of recomputing SCCs, their sparse certificates and G' before any sequence of edge deletions, SCCs are merged to supervertices as soon as they are created. This way, the recomputation of the data structure before each sequence of edge deletions is avoided (thus speeding up mixed sequences of operations). This further allows one to adopt the lazy approach for resetting the data structure as applied in `Ital-Opt`. This concludes the description of Italiano's algorithm and its variants.

Yellin's data structure associates with every vertex v the doubly linked list *Adjacent*(v) of the heads of its outgoing edges, and the doubly linked list *Reaches*(v) of the tails of its incoming edges. In addition, an $n \times n$ array *INDEX* is maintained. Each entry *INDEX*$[v, w]$ has (among others) a field called *refcount* that stores the number of v-w paths in G, defined as *refcount*$(v, w) = |ref(v, w)| + 1$, if $(v, w) \in E$, and *refcount*$(v, w) = |ref(v, w)|$, otherwise, where $ref(v, w) = \{(v, z, w) : z \in V \wedge (v, z) \in E^* \wedge (z, w) \in E\}$.

The main idea for updating the data structure after the insertion of an edge (a, b) is to find, for all $x, z \in V$, the new triples (x, y, z) that should be added to $ref(x, z)$ and update *INDEX*$[v, w]$.*refcount*. The insertion algorithm first finds all vertices x such that (x, a) was an edge of G^*_{old} (the transitive closure graph before the insertion of (a, b)). In this case, (x, a, b) is a new triple in $ref(x, b)$, and *refcount*(x, b) has to be incremented. Then, the insertion algorithm considers each new edge (x, y) in G^*_{new} (the transitive closure graph after the insertion of (a, b)) and each edge (y, z) of G; (x, y) is a new transitive closure edge if its *refcount* was increased from 0 to 1. Now, (x, y, z) is a new triple for $ref(x, z)$ and *refcount*(x, z) is increased by 1.

The edge deletion algorithm is the "dual" of the edge insertion algorithm described above.

To support `Path` queries, Yellin's algorithm has to be augmented with a rather "heavy" data structure called the *support graph*, which stores all transitive closure edges and all triples (x, y, z) in $ref(x, z)$. Consequently, it occupies $O(n^3)$ space, making this version of Yellin's algorithm very slow and space consuming, a fact that was indeed confirmed by the experiments in [11.32, 11.33]. It is also easy to see that the data structures in either version need no modification in order to be used in a fully dynamic environment.

The algorithm of Cicerone et al. [11.12] provides a uniform approach for maintaining several binary relationships (e.g., transitive closure, dominance, transitive reduction) incrementally on general digraphs and decrementally on DAGs. The main advantage of this technique, besides its simplicity, is the fact that its implementation does not depend on the particular problem; i.e., the same procedures can be used to deal with different problems by simply setting appropriate boundary conditions.

The approach allows a *propagation property* to be defined based on a binary relationship (e.g., transitive closure) $R \subseteq V \times V$ that describes how R "propagates" along the edges of a digraph $G = (V, E)$. More precisely, a relationship R satisfies the propagation property over G with boundary condition $R_0 \subset R$ if, for any pair $\langle x, y\rangle \in V \times V$, $\langle x, y\rangle \in R$ if and only if either $\langle x, y\rangle \in R_0$, or $x \neq y$ and there exists a vertex $z \neq y$ such that $\langle x, z\rangle \in R$ and $(z, y) \in E$. The relationship R_0 is used to define the set of elements of R that cannot be deduced using the propagation property. For example, if R is the transitive closure, then $R_0 = \{(x, x) : x \in V\}$. Actually, if R is the transitive closure, then the algorithm of [11.12] collapses to the algorithm of La Poutré and van Leeuwen [11.49].

The data structure maintains an integer matrix that contains, for each pair of vertices $\langle x, y\rangle \in V \times V$, the number $U_R[x, y]$ of edges useful to that pair. An edge $(z, y) \in E$ is *useful* to pair $\langle x, y\rangle$ if $z \neq y$ and $\langle x, z\rangle \in R$. Now, it is easy to see that, for any pair $\langle x, y\rangle \in V \times V$, $\langle x, y\rangle \in R$ if and only if either $\langle x, y\rangle \in R_0$ or $U_R[x, y] > 0$. In addition to the $n \times n$ integer matrix described above, and the binary matrix representing the boundary condition R_0, two additional data structures are maintained: (a) a set $\text{OUT}[x]$, for each vertex x, that contains all outgoing edges of x; and (b) a queue Q_k, for every vertex k, to handle edges (h, y) useful to pair $\langle k, y\rangle$. A `Boolean` query for vertices i and j takes $O(1)$ time, since it involves only checking the value $U_R[i, j]$.

After the insertion of edge (i, j) the number of edges useful to any pair $\langle k, y\rangle$ can only increase. An edge insertion is performed as follows: firstly, for each vertex k, the new edge (i, j) is inserted into the empty queue Q_k if and only if $\langle k, i\rangle \in R$, and hence it is useful to pair $\langle k, j\rangle$; then, edges (t, h) are extracted from queues Q_k, and the values $U_R[k, h]$ are increased by one, because these edges are useful to pair (k, h). Now, edges $(h, y) \in \text{OUT}[h]$ are inserted in Q_k if and only if the pair $\langle k, h\rangle$ has been added for the first time

to R as a consequence of an edge insertion, i.e., if and only if $U_R[k,h] = 1$. This implies that, during a sequence of edge insertions, the edge (h,y) can be inserted in Q_k at most once.

The behaviour of an edge deletion operation is analogous. After the deletion of edge (i,j) some edges could no longer be useful to a pair $\langle k,y\rangle$, and then the corresponding value $U_R[k,y]$ has to be properly decreased. An edge deletion is handled as follows: firstly, for each vertex k, the deleted edge (i,j) is inserted into the empty queue Q_k if and only if $\langle k,i\rangle \in R$; then, edges (t,h) are extracted from queues Q_k, and the values $U_R[k,h]$ are decreased each time by one, because these edges are no longer useful to pair (k,h). Now, edges $(h,y) \in \text{OUT}[h]$ are inserted in Q_k if and only if $U_R[k,h] = 0$ and $(k,h) \notin R_0$, that is, when there is no edge useful to pair (k,h). This implies that, during a sequence of edge deletions, the edge (h,y) can be inserted in Q_k at most once. Notice that, if $U_R[k,h] > 0$, then (k,h) is still in R after deleting edge (i,j), because G is acyclic.

Two different variants of the algorithms in [11.12] were implemented in [11.32, 11.33]: the general technique, denoted as `CFNP`, and its specialization to the transitive closure problem, denoted as `CFNP-Opt`. The main difference between the two implementations is that after each edge insertion, the original algorithm (`CFNP`) performs at least a computation of $O(n)$ time in order to update the counters modified by that insertion; on the other hand, after an edge insertion `CFNP-Opt` starts its computation only when the inserted edge (i,j) introduces a new path between i and j (an idea borrowed from Italiano's approach). Thus, instead of the matrix of counters, `CFNP-Opt` simply maintains a binary matrix representing the transitive closure of the graph.

As with Yellin's algorithm, `CFNP` can be used in a fully dynamic environment without any modification on its data structure. On the other hand, `CFNP-Opt` cannot be used in such an environment.

The Henzinger-King algorithms [11.37] are based on the maintenance of BFS trees of vertices reachable from (or which reach) a specific distinguished vertex, and the fact that with very high probability every vertex in the graph reaches (or is reachable by) a distinguished vertex by a path of small distance (counted in number of edges).

Let $out(x,k)$ (resp. $in(x,k)$) denote the set of vertices reachable from (resp. which reach) vertex x by a path of distance at most k. The decremental algorithm, denoted as `HK-1`, selects at random sets of distinguished vertices S_i, for $i = 1,\ldots,\log n$, where $|S_i| = \min\{O(2^i \log n), n\}$. For every $x \in S_i$ the algorithm maintains (a) $out(x, n/2^i)$ and $in(x, n/2^i)$; and (b) $Out(x) = \bigcup_{i:x\in S_i} out(x, n/2^i)$ and $In(x) = \bigcup_{i:x\in S_i} in(x, n/2^i)$. In addition, for each $u \in V$ the sets $out(u, \log^2 n)$ and $in(u, \log^2 n)$ are maintained. The sets $out(x,k)$ and $in(x,k)$ are maintained in a decremental environment using a (modification of a) technique proposed by Even and Shiloach [11.21] for undirected graphs. Each set is called a *BFS structure*, since it implicitly maintains a spanning tree for the descendants of x as it would have been computed

by a BFS algorithm. Hence, an edge deletion reduces to the maintenance of certain BFS structures.

A query for vertices u and v is carried out as follows. Check if v is in $out(u, \log^2 n)$. If not, then check for any vertex $x \in S$ whether $u \in In(x)$ and $v \in Out(x)$. If such an x exists, then there is a u-v path; otherwise, such a path does not exist with high probability. A `Boolean` query is answered in time proportional to $|S|$, i.e., in $O(n/\log n)$ time, and a `Path` query is answered in an additional $O(\ell)$ time, where ℓ is the length of the reported path.

The `HK-1` implementation in [11.32, 11.33] was further "engineered" to improve performance, by reducing the required space and the time to check whether there is a x-u (resp. u-x) path in an ($out(x,k)$ (resp. $in(x,k)$) set. The former is accomplished by keeping in $Out(x)$ and $In(x)$ only the tree with the largest depth among the potentially many trees having as root the same vertex. The latter is accomplished in $O(1)$ time by assigning to the vertices not in such a set a level greater than k.

The fully dynamic algorithm, denoted as `HK-2`, keeps the above decremental data structure to give answers if there is an "old" path between two vertices (i.e., a path that does not use any of the newly inserted edges). Updates are carried out as follows. After the insertion of an edge (i,j), compute $in(i,n)$ and $out(i,n)$. After the deletion of an edge, recompute $in(i,n)$ and $out(i,n)$ for all inserted edges (i,j), and update the decremental data structure for old paths. Rebuild the decremental data structure after $\sqrt{n}$ updates. To answer a query for u and v, check first if there is an old path between them. If not, then check if $u \in in(i,n)$ and $v \in out(i,n)$ for all i which are tails of the newly inserted edges (i,j).

Finally, three simple-minded (pseudo-dynamic) algorithms were implemented in [11.32, 11.33] and compared to the above dynamic algorithms; they were based on the following method: in the case of an edge insertion (resp. deletion), the new (resp. existing) edge is simply added to (resp. removed from) the graph and nothing else is computed. In the case of a query, a search from the source vertex s is performed until the target vertex t is reached (if an s-t path exists) or until all vertices reachable from s are exhausted. Depending on the search method used (DFS, BFS, and a combination of them), the three different implementations require no initialization time, $O(1)$ time per edge insertion or deletion, and $O(n+m)$ time per query operation, where m is the current number of edges in the graph. The implementation of the simple-minded algorithms include: `DFS`, `BFS`, and `DBFS`. The latter is a combination of `DFS` and `BFS` that works as follows. Vertices are visited in DFS order. Every time a vertex is visited, first check whether any of its adjacent vertices is the target vertex. If yes, then stop; otherwise, continue the visit in a DFS manner. The experimental results in [11.32, 11.33] showed that there were cases where `DBFS` outperformed `DFS` and `BFS`.

An extensive experimental study of this bulk of implementations was conducted in [11.32, 11.33]. The experiments were run on three different kinds of inputs: random inputs (aimed at identifying the average-case performance of the algorithms), non-random inputs (aimed at forcing the algorithms to exhibit their worst-case performance), and on a more pragmatic input motivated by a real world graph: the graph describing the connections among the autonomous systems of a fragment of the Internet network visible from *RIPE* (`www.ripe.net`), one of the main European servers. On this graph random sequences of operations were performed.

Random inputs consisted of randomly generated digraphs and DAGs with different edge densities ($m_0 \in \{0, n/2, n, n \ln n, n^{1.5}, n^2/\ln n, n^2/4\}$) and random sequences of operations. As it is mentioned in Section 11.2.1.2.a, the values $n/2, n, n \ln n$ are thresholds around which a fundamentally different structural behaviour of a random graph occurs [11.9] (the rest of the values are chosen as intermediate steps towards denser graphs). In the random sequences of operations, the queries were uniformly mixed with updates (insertions/deletions). Moreover, various lengths of such sequences were considered.

The non-random inputs were inspired from those proposed in [11.3] (cf. Section 11.2.1.2.a). They consisted of a number k of cliques (complete digraphs or complete DAGs), each one containing roughly n/k vertices, and which are interconnected by a set B of at most $k-1$ inter-clique edges, called "bridges". Depending on the type of updates, the edges in B are precisely those which are inserted or deleted from the graph during a sequence of operations. This forces the dynamic algorithms to handle dense subgraphs while the reachability information of the whole graph keeps changing. To make it even harder (and assuming an arbitrary numbering of the cliques), there is a specific order in which these edges were inserted (resp. deleted) in an incremental (resp. decremental) environment. In the incremental case, the graph G initially has no bridges. Bridges are added from B as follows: the first bridge is inserted between the first and the second clique, the second bridge between the penultimate and the ultimate clique, the third bridge between the second and the third clique, and so on. Hence, the bridge inserted last will provide new reachability information from roughly $n/2$ to the other $n/2$ vertices of G. The reverse order is followed in the case of edge deletions, where all edges from B were initially in G.

For random inputs, the reported experiments were as follows. In the incremental case as well as in the decremental case for DAGs, `Ital-Opt` and `Ital-Gen` were almost always the fastest. The simple-minded algorithms became competitive or faster in: (i) the incremental case when the initial graph was very sparse (containing less than $n/2$ edges); (ii) the decremental case when the initial graph was sparse (containing less than $n \ln n$ edges) and the sequence of operations was of medium to small size. This could be explained by the fact that as the graph becomes denser edge insertions do not

add new information w.r.t. transitive closure, while edge deletions below the connectivity threshold ($n \ln n$ for random digraphs) increase considerably the work performed by the dynamic algorithms (e.g., Italiano's algorithms have greater difficulty to find hooks in this case) which cannot be amortized with the length of the operation sequence. In the decremental case for general digraphs, the simple-minded algorithms were always significantly faster than `HK-1` or `Ital-Gen`, because the former suffered by the updating of the BFS structures, while the latter suffered by the splitting of SCCs and the recomputation of their sparse certificates. Similar behaviour was observed in the fully dynamic case for general digraphs; an interesting fact was that `HK-2` was the slowest in practice, even for very small sequences of operations where the algorithm is assumed to perform well (i.e., does not perform a rebuild). In the fully dynamic case for DAGs, again `Ital-Opt` and `Ital-Gen` were the fastest when the initial graph was not sparse (having more than $n \ln n$ edges); in the sparse case, the simple-minded algorithms became competitive. The efficiency of `Ital-Opt` and `Ital-Gen` demonstrates that the lazy approach for resetting the hooks was indeed successful.

In the case of non-random inputs, the simple-minded algorithms were significantly faster than any of the dynamic algorithms. The best dynamic algorithms were `Ital-Opt` or `Ital-Gen`. An interesting observation for this type of inputs concerned the `HK-2` algorithm: for small values of k, it was the slowest. For larger values of k, it became competitive or faster than the best dynamic (`Ital-Opt` or `Ital-Gen`). This behaviour is due to the rebuilding of its data structure after a certain threshold in the length of the operation sequence. The larger the value of k, the less rebuilds are performed by `HK-2`.

Finally, the experiments with the fragment of the Internet graph gave similar conclusions to those obtained with random inputs. Additional experiments have been performed with operation sequences for which some knowledge about their update-query pattern is known in advance (e.g., the percentage of queries). Although the theoretical bounds of the dynamic algorithms may not hold in this case, these experiments might give useful suggestions on how to proceed if information about the update-query pattern is provided. The experimental results provided a quantitative idea of what is the break point, i.e., after which percentage of queries the dynamic algorithms overcome the simple-minded ones. As expected, the higher the percentage of queries, the worse the simple-minded algorithms. The break points, however, differ in each dynamic setting.

The source code of the above implementations is available from `http://www.ceid.upatras.gr/faculty/zaro/software`.

11.3.1.2.b *The Implementation by Abdeddaim.* [11.2]

The main goal of that paper was to investigate whether certain knowledge about the input digraph (in terms of the so-called "path cover") could help to speed up the computation time for maintaining the transitive closure in an incremental environment (edge insertions and queries). The motiva-

tion behind this interest lies on the fact that incremental transitive closure is a fundamental subroutine in algorithms for sequence alignment and that alignment graphs have the requested knowledge.

In [11.2], the incremental algorithms developed in [11.1, 11.2] were implemented and compared with implementations of Italiano's incremental algorithm [11.42] which appears to be among the fastest in the previous study (cf. Section 11.3.1.2.a).

The algorithms in [11.1, 11.2] assume that the input digraph is provided with a set of k vertex-disjoint paths that cover all vertices of the graph. Such a set is called a *path cover*. Finding a path cover of minimum cardinality is an NP-complete problem (see e.g., [11.45]). However, there are families of graphs for which a path cover (not necessarily minimum) can either be efficiently computed or is known in advance. The former case includes DAGs, where the computation of a minimum path cover reduces to a minimum flow problem [11.45]. The latter case includes alignment graphs which are used in greedy algorithms for sequence alignment [11.1]. Since computing a minimum flow takes roughly $O(nm)$ time (as it reduces to the max-flow problem), one can resort to other approaches to find a path cover in a DAG which may not be minimum. Such an algorithm is described in [11.51, pp. 11–12] and in [11.60], and takes linear time. This idea can be extended to general digraphs by shrinking each strongly connected component to a single vertex, thus yielding a DAG on which the algorithm of [11.51, 11.60] can be applied.

The main idea of the algorithms in [11.1, 11.2] is to use the initial path cover to compute, for each vertex x, a *predecessor* and a *successor frontier* which encode the number of predecessors and successors of x in each path, respectively. When a new edge (x, y) is inserted, it is first checked whether there is already an x-y path in the graph. If yes, then the frontiers are not modified. Otherwise, the algorithms consider each pair of paths P_i and P_j, and for each predecessor u of x in P_i, its successors' number in P_j is updated by the maximum of its current value and the number of successors of y in P_j. The predecessor frontiers are computed in a similar way. The difference between the two algorithms in [11.1, 11.2] lies in the order in which u and j are considered. The algorithm from [11.1] (resp. [11.2]) is referred to as `Abd97` (resp. `Abd99`).

In the experimental study of [11.2] two kind of inputs were considered: random inputs and alignment graphs.

The random inputs are different from those we have seen so far. They are constructed as follows: firstly, k paths of random lengths were generated by assigning to each vertex v a random label l_v in $[1, k]$ and considering vertices with the same label to belong to the same path. Secondly, a set of edges chosen uniformly at random were added.

The alignment graphs were taken from two benchmarks libraries, namely the BAliBASE [11.63] and the PFAM [11.8].

The Abd97 and Abd99 algorithms were implemented in ANSI C and compared to an implementation, called Ita86 (also in ANSI C), of the original incremental algorithm by Italiano [11.42] that was developed in [11.2], and to the Ital-Opt and CFNP-Opt implementations from [11.32]. Note that the latter two are C++ implementations using LEDA [11.52] and hence expected to be slower than an ANSI C implementation. From the theoretical analysis of Abd97 and Abd99 (cf. Section 11.3.1.1), it turns out that for small values of k and sufficiently large operation sequences both algorithms achieve a very good amortized time per update operation. This was exactly confirmed by the experiments in [11.2]. For both types of inputs, Abd97 and Abd99 were faster than the other algorithms, with Abd97 being usually the fastest. When the value of k was getting larger, or the graph was becoming denser, Abd99 was taking over.

It would be interesting to investigate the performance of Abd97 and Abd99 on the data sets developed in [11.32, 11.33].

11.3.1.3 Lessons Learned. The above experimental studies allow us to obtain a better understanding of existing dynamic algorithms for transitive closure and their practical assessment, and also to detect certain differences in their practical behaviour compared with that of their undirected counterparts for dynamic connectivity. In particular:

- In partially dynamic environments on unstructured inputs, dynamic algorithms are usually faster than simpler approaches (based on static algorithms). On the other hand, the simpler approaches are considerably better than dynamic algorithms either in fully dynamic environments on unstructured inputs, or in any kind of dynamic environment on structured inputs.
- To beat the simpler approaches, thorough experimentation, fine-tuning, and sophisticated engineering of dynamic algorithms is required.
- Certain knowledge about the input digraph or the update sequence turns out to be useful in practice.
- The test suite of [11.32, 11.33], generated by a methodology analogous to that in [11.3, 11.4], along with the enhancement of more pragmatic inputs (e.g., inputs from benchmark libraries, real-world inputs, etc) can be considered as a valuable benchmark for testing other dynamic algorithms for directed graphs.

11.3.2 Dynamic Shortest Paths

11.3.2.1 Theoretical Background — Problem and History of Results. The shortest path problem consists in finding paths of minimum total weight between specified pairs of vertices in a digraph $G = (V, E)$ whose edges are associated with real-valued weights. A path of minimum total weight between two vertices x and y is called *shortest path*; the weight of a shortest x-y path is called the *distance* from x to y. There are two main versions of

the shortest path problem: the *all-pairs shortest paths* (APSP) in which we seek shortest paths between every pair of vertices in G; and the *single-source shortest path* (SSSP) in which we seek shortest paths from a specific vertex s to all other vertices in G.

The *dynamic shortest path* problem consists in building a data structure that supports query and update operations. A shortest path (resp. distance) query specifies two vertices and asks for the shortest path (resp. distance) between them. An update operation updates the data structure after an edge insertion or edge deletion or edge weight modification. There are several algorithms for both the dynamic APSP and the dynamic SSSP problems. Actually, dynamic shortest path problems have been studied since 1967 [11.50, 11.54, 11.58]. In the following, let $n = |V|$ and let m_0 denote the initial number of edges in G.

For the dynamic APSP problem and the case of arbitrary real-valued edge weights, Even & Gazit [11.20] and Rohnert [11.59] gave (independently) two fully dynamic algorithms in 1985. Both algorithms create a data structure which is initialized in $O(nm_0 + n^2 \log n)$ time, supports shortest path or distance queries optimally, and is updated either in $O(n^2)$ time after an edge insertion or edge weight decrease, or in $O(nm + n^2 \log n)$ time after an edge deletion or edge weight increase (m being the current number of edges in the graph). These were considered the best algorithms for dynamic APSP on general digraphs with arbitrary real-valued edge weights, until a very recent breakthrough achieved by Demetrescu & Italiano [11.14]: if each edge weight can assume at most S different real values, then any update operation can be accomplished deterministically in $O(Sn^{2.5} \log^3 n)$ amortized time and a distance query in $O(1)$ time. In the same paper, an incremental randomized algorithm (with one-sided error) is given which supports an update in $O(Sn \log^3 n)$ amortized time.

In the case where the edge weights are nonnegative integers, a number of results were known. Let C denote the largest (integer) value of an edge weight. In [11.7], Ausiello et al. gave an incremental algorithm that supports queries optimally, and updates its data structure in $O(Cn^3 \log(nC))$ time after a sequence of at most $O(n^2)$ edge insertions or at most $O(Cn^2)$ edge weight decreases. More recently, a fully dynamic algorithm was given by King [11.47] which supports queries optimally, and updates (edge insertions or deletions) in $O(n^{2.5}\sqrt{C \log n})$ amortized time (amortized over a sequence of operations of length $\Omega(m_0/n)$). Also, in the same paper a decremental algorithm is given which supports any number of edge deletions in $O(m_0 n^2 C)$ time (i.e., in $O(n^2 C)$ amortized time per deletion if there are $\Omega(m_0)$ deletions). We note that very efficient dynamic algorithms are known for special classes of digraphs (planar, outerplanar, digraphs of small treewidth, and digraphs of small genus) with arbitrary edge weights; see [11.11, 11.17].

The efficient solution of the dynamic SSSP problem is a more difficult task, since almost optimal algorithms are known for the static version of

the problem. Nothing better than recomputing from scratch is known about the dynamic SSSP problem, with the exception of a decremental algorithm presented in [11.24]. That algorithm assumes integral edge weights in $[1..C]$ and supports any sequence of edge deletions in $O(m_0 nC)$ time (i.e., $O(nC)$ amortized time per deletion, if there are $\Omega(m_0)$ deletions).

For the above and other reasons most of the research for the dynamic SSSP problem has been concentrated on different computational models. One such model is the *output complexity cost model* introduced by Ramalingam & Reps [11.56, 11.57] and extended by Frigioni et al. in [11.29, 11.31]. In this model, the time-cost of a dynamic algorithm is measured as a function of the number of updates to the output information of the problem caused by input updates.

Let δ denote an input update (edge insertion, edge deletion, or edge weight modification) to be performed on the given digraph G, and let V_δ be the set of *affected vertices*, i.e., the vertices that change their output value as a consequence of δ (e.g., for the SSSP problem their distance from the source s). In [11.56, 11.57], the cost of a dynamic algorithm is measured in terms of the *extended size* $\|\delta\|$ of the change in input and output. Parameter $\|\delta\|$ equals the sum of $|V_\delta|$ and the number of edges that have at least one affected endpoint. Note that $\|\delta\|$ can be $O(m)$ in the worst-case, and that both $\|\delta\|$ and $|V_\delta|$ depend only on the problem instance. In [11.56, 11.57] a fully dynamic algorithm is provided that updates its data structure after a change δ in the input in time $O(\|\delta\| + |V_\delta| \log |V_\delta|)$. Queries are answered optimally.

In [11.29], the cost of a dynamic algorithm is measured in terms of the number of changes on the *output information* of the problem. In the case of the SSSP problem, the output information is the distances of the vertices from s and the shortest path tree. The output complexity cost is measured in this case as a function of the *number of output updates* $|U_\delta|$, where U_δ (the set of output updates) consists of those vertices which, as a consequence of δ, either change their distance from s, or must change their parent in the shortest path tree (even if they maintain the same distance). Differently from the model of [11.56, 11.57], U_δ depends on the current shortest path tree (i.e., on the algorithm used to produce it.) In [11.29] an incremental algorithm for the dynamic SSSP is given which supports queries optimally and updates its data structure in time $O(k|U_\delta| \log n)$ where k is a parameter bounded by structural properties of the graph. For general graphs $k = O(\sqrt{m})$, but for special classes of graphs it can be smaller (e.g., in planar graphs $k \leq 3$). A fully dynamic algorithm with the same update and query bounds is achieved in [11.31].

All the above algorithms [11.29, 11.31, 11.56, 11.57] for the dynamic SSSP problem require that the edge weights are nonnegative and that there are no zero-weighted cycles in the graph either before or after an input update δ. These restrictions have been waived in [11.30] where a fully dynamic algorithm is presented that answers queries optimally and updates its data struc-

ture in $O(\min\{m, kn_a\} \log n)$ time after an edge weight decrease (or edge insertion), and in $O(\min\{m \log n, k(n_a + n_b) \log n + n_c\})$ time after an edge weight increase (or edge deletion). Here, n_a is the number of affected vertices, n_b is the number of vertices that preserve their distance from s but change their parent in the shortest path tree, and n_c is the number of vertices that preserve both their distance from s and their parent in the shortest path tree.

11.3.2.2 Implementations and Experimental Studies. There are two works known regarding implementation and experimental studies of dynamic algorithms for shortest paths. In chronological order, these are the works by Frigioni et al. [11.28] and by Demetrescu et al. [11.15].

Both studies deal with the dynamic SSSP problem and aim at identifying the practicality of dynamic algorithms over static approaches. The former paper investigates the case of non-negative edge weights, while the latter investigates the case of arbitrary edge weights.

11.3.2.2.a *The Implementation by Frigioni et al.* [11.28]

The main goal of the first experimental study on dynamic shortest paths was to investigate the practicality of fully dynamic algorithms over static ones, and to experimentally validate the usefulness of the output complexity model.

In particular, the fully dynamic algorithms by Ramalingam & Reps [11.56] and by Frigioni et al. [11.31] have been implemented and compared with a simple-minded, pseudo-dynamic algorithm based on Dijkstra's algorithm. Both fully dynamic algorithms are also based on suitable modifications of Dijkstra's algorithm [11.16]. Their difference lies in how the outgoing edges of a vertex are processed when its distance from s changes. In the following, let $d(v)$ denote the current distance of a vertex v from s and let $c(u, v)$ denote the weight of edge (u, v).

The algorithm of Ramalingam & Reps [11.56] maintains a DAG $SP(G)$ containing all vertices of the input graph G and exactly those edges that belong to at least one shortest path from s to all other vertices of G.

In the case of an edge insertion, the algorithm proceeds in a Dijkstra-like manner on the vertices affected by the insertion. Let (v, w) be the inserted edge. The algorithm stores the vertices in a priority queue Q with priority equal to their distance from w. When a vertex x of minimum priority is deleted from Q, then all of its outgoing edges (x, y) are traversed. A vertex y is inserted in Q, or its priority is updated, if $d(x) + c(x, y) < d(y)$. In such a case, (x, y) is added to $SP(G)$ and all incoming edges of y are deleted from $SP(G)$. If $d(x) + c(x, y) = d(y)$, then (x, y) is simply added to $SP(G)$.

In the case of an edge deletion, the algorithm proceeds in two phases. Let A (resp. B) denote the set of affected (resp. unaffected) vertices. In the first phase the set A of affected vertices is determined by performing a kind of topological sorting on $SP(G)$. Let (v, w) be the deleted edge. Vertex w is put into A, if its indegree in $SP(G)$ is zero after the deletion of (v, w). If $w \in A$, then all of its outgoing edges are deleted from $SP(G)$. If this yields new

vertices of zero indegree, then they added to A, and their outgoing edges are deleted from $SP(G)$. The process is repeated until all vertices are exhausted or there are no more vertices of zero indegree in $SP(G)$. In the second phase, the new distances of the vertices in A are determined. This is done by first shrinking the subgraph induced by B on a new super-source s', and then by adding, for each edge (x, y) with $x \in B$ and $y \in A$, the edge (s', y) with weight equal to $d(x) + c(x, y)$. Finally, run Dijkstra's algorithm with source s' on the resulting graph and update $SP(G)$ as in the case of edge insertion. The implementation of the above algorithm is referred to as `RR` in [11.28].

The algorithm by Frigioni et al. [11.31] is based on a similar idea, but an additional data structure is maintained on each vertex v in order to "guess" which neighbors of v have to be updated when the distance from v to the source changes. This data structure is based on the notions of the *level* and the *owner* of an edge. The *backward level* of an edge (y, z), associated with vertex z, is defined as $BL_y(z) = d(z) - c(y, z)$; the *forward level* of an edge (x, y), associated with vertex x, is defined as $FL_y(x) = d(x) + c(x, y)$. Intuitively, these levels provide information about the shortest available path from s to z that passes through y. The *owner* of an edge is one of its two endpoints, but not both. The incoming and outgoing edges of a vertex y is partitioned into those owned by y and into those not owned by y. The incoming and outgoing edges not owned by y are stored in two priority queues F_y (for the incoming) and B_y (for the outgoing) with priorities determined by their forward and backward levels, respectively. Any time a vertex y changes its distance from s, the algorithm traverses all edges owned by y and an appropriately chosen subset of edges not owned by y.

When the insertion of an edge (v, w) decreases $d(w)$, then a priority queue Q' is used, as in Dijkstra's algorithm, to find new distances from s. However, differently from Dijkstra's and the `RR` algorithm, when a vertex y is deleted from Q' and its new distance decreases, only those not-owned outgoing edges (y, z) are scanned whose priority in B_y is greater than the new $d(y)$, since only in this case (i.e., $BL_y(z) = d(z) - c(y, z) > d(y)$) $d(z)$ is decreased by a shortest s-z path that passes through y.

In the case of an edge deletion, the algorithm proceeds in two phases (like `RR`). In the first phase, the affected vertices are determined. When a vertex y increases its distance due to the edge deletion, then in order to find the best possible alternative shortest s-y path, only those not-owned incoming edges (x, y) are scanned whose priority in F_y is smaller than $d(y)$. The vertex x which minimizes $FL_y(x) - d(y)$ is the new parent of y in the shortest path tree. The increase in $d(y)$ is propagated to the outgoing edges owned by y. In the second phase, the distances of the affected vertices are computed by performing a Dijkstra-like computation on the subgraph induced by those vertices and by considering only edges between affected vertices. The implementation of the above algorithm is referred to as `FMN` in [11.28].

Finally, a pseudo-dynamic algorithm, called `Dij`, was implemented based on LEDA's implementation of Dijkstra's algorithm. It simply recomputes from scratch the shortest path information, only when an input update affects this information. Since Dijkstra's implementation in LEDA uses Fibonacci heaps, all priority queues implemented in `RR` and `FMN` are also Fibonacci heaps.

The implementations `RR`, `FMN`, and `Dij` were experimentally compared in [11.28] using three kinds of inputs: random inputs, structured inputs, and on the graph describing the connections among the autonomous systems of a fragment of the Internet network visible from *RIPE* (`www.ripe.net`), one of the main European servers.

For random inputs, two types of operation sequences were performed on random graphs: randomly generated sequences of updates, and modifying sequences of updates. In the latter type, an operation is selected uniformly at random among those which actually modify some shortest path from the source. Edge weights are chosen randomly.

The structured input consisted of a special graph and a specific update sequence on that graph. The graph consists of a source s, a sink t, and a set X of k other vertices $x_1, \ldots, x_k$. The edge set consists of edges (s, x_i), (x_i, s), (t, x_i) and (x_i, t), for $1 \leq i \leq k$. The sequence of updates consists of alternated insertions and deletions of the single edge (s, t) with a proper edge weight. The motivation for this input was to exhibit experimentally the difference between the complexity parameters $\|\delta\|$ used by `RR` and $|U_\delta|$ used by `FMN`, since the theoretical models proposed by [11.56] and [11.29] are different and do not allow for a direct comparison of these parameters. Clearly, with this input after each dynamic operation only the distance of t changes. Hence, it is expected that as the size of the neighborhood of the affected vertices increases, `FMN` should dominate over `RR`: after the insertion (resp. deletion) of (s, t), `RR` visits all k edges outgoing from (resp. incoming to) t, while `FMN` visits only those edges "owned" by t.

The input based on the fragment of the Internet graph consists of unary weights and random update sequences.

In all experiments performed with any kind of input, both `RR` and `FMN` were substantially faster than `Dij` (although in the worst-case the bounds of all algorithms are identical). In the case of random inputs, `RR` was faster than `FMN` regardless of the type of the operation sequence. In the cases of structured input and of the input with the fragment of the Internet graph, `FMN` was better. An interesting observation was that on any kind of input, the edges scanned by `FMN` were much less than those scanned by `RR` (as expected). However, `FMN` uses more complex data structures which, in the case of random inputs, eliminate this advantage.

The source code of the above implementations is available from `http://www.jea.acm.org/1998/FrigioniDynamic`.

11.3.2.2.b *The Implementation by Demetrescu et al.* [11.15]

The main goal of that paper was to investigate the practical performance of fully dynamic algorithms for the SSSP problem in the case of digraphs with arbitrary edge weights.

In particular, the algorithms considered were the recent fully dynamic algorithm by Frigioni et al. [11.30], referred to as `FMN-gen`; a simplified version of it, called `DFMN`; a variant of the `RR` algorithm, referred to as `RR-gen` [11.57] which works with arbitrary edge weights; and a new simple dynamic algorithm, referred to as `DF`.

The common idea behind all these algorithms is to use the Edmonds-Karp technique [11.18] to transform an SSSP problem with arbitrary edge weights to another one with nonnegative edge weights without changing the shortest paths. This is done by replacing each edge weight $c(x, y)$ by its reduced version $r(x, y) = d(x) - d(y) + c(x, y)$ (the distances $d(\cdot)$ are provided by the input shortest path tree), running Dijkstra's algorithm to the graph with the reduced edge weights (which are nonnegative), and then trivially obtain the actual distances from those based on the reduced weights.

In the case of an edge insertion or weight decrease operation, `FMN-gen` and `DFMN` behave similarly to `FMN` (cf. Section 11.3.2.2.a), while `DF` and `RR-gen` behave similarly to `RR` (cf. Section 11.3.2.2.a). However, `DF` has not been designed to be efficient according to the output complexity model as `RR` had, and its worst-case complexity is $O(m + n \log n)$.

In the case of an edge deletion or weight increase operation, there are differences in the algorithms. Algorithm `FMN-gen` proceeds similarly to `FMN` (cf. Section 11.3.2.2.a), but the traversal of the not-owned incoming edges becomes more complicated as zero-weighted cycles should be handled. The `DFMN` algorithm is basically the `FMN-gen` algorithm without the partition of the incoming and outgoing edges into owned and not-owned. This allows for simpler and, as experiments showed, faster code. Finally, the `DF` algorithm, as in the case of edge insertion or weight decrease operation, uses the classical complexity model and not the output complexity one, and its worst-case complexity is $O(m+n \log n)$. Let (x, y) be the edge whose weight is increased by a positive amount Δ. The algorithm consists of two phases, called *initializing* and *updating*. In the initializing phase, all vertices in the subtree $T(y)$ of the shortest path tree rooted at y are marked. Each marked vertex v finds its "best" unmarked neighbor u in its list of incoming edges. This yields a (not necessarily shortest) s-v path whose weight, however, is used as the initial priority of v (i.e., of an affected vertex) in a priority queue H used in the updating phase. If u is not `nil` and $d(u) + c(u, v) - d(v) < \Delta$, then the priority of v equals $d(u)+c(u, v)-d(v)$; otherwise, it equals Δ. In either case, the initial priority is an upper bound on the actual distance. The updating phase simply runs Dijkstra's algorithms on the marked vertices inserted in H with the above initial priorities.

The experiments in [11.15] were conducted only on random inputs. In particular, they were performed on randomly generated digraphs and various update sequences, which enhance in several ways the random inputs considered in [11.28] (cf. Section 11.3.2.2.a).

Random digraphs were generated such that all vertices are reachable from the source and edge weights are randomly selected from a predetermined interval. The random digraphs come in two variants: those forming no negative or zero weight cycles, and those in which all cycles have weight zero.

The update sequences were random update sequences (uniformly mixed sequences of edge increase and decrease operations that do not introduce negative or zero weight cycles), modifying update sequences (an operation is selected uniformly at random among those which actually modify some shortest path from the source), and alternate update sequences (updates alternate between edge weight increase and decrease operations and each consecutive pair of increase-decrease operation is performed on the same edge).

In all experiments, `FMN-gen` was substantially slower than `DFMN`, since it uses more complex data structures. In the experiments with arbitrary edge weights, but no zero-weighted cycles, `DF` was the fastest algorithm followed by `RR-gen`; `DFMN` is penalized by its additional effort to identify affected vertices in a graph that may have zero-weighted cycles. It is interesting to observe that `RR-gen` is slightly faster than `DF` when the range of values of the edge weights is small. In the case of inputs which included zero-weighted cycles, either in the initial graph or because of a specific update sequence which tried to force cycles in the graph to have weight zero, `DFMN` outperformed `DF`. Note that in this case `RR-gen` is not applicable.

The source code of the above implementations is available from `ftp://www.dis.uniroma1.it/pub/demetres/experim/dsplib-1.1`.

11.3.2.3 Lessons Learned. The experimental studies in [11.28] and [11.15] enhance our knowledge on the practicality of several algorithms for the dynamic SSSP problem. In particular:

- The output cost model is not only theoretically interesting, but appears to be quite useful in practice.
- Fully dynamic algorithms for the SSSP problem compare favorably in practice to almost optimal static approaches.
- The random test suite developed initially in [11.28] and considerably expanded and elaborated in [11.15] provides an important benchmark of random inputs for future experimental studies with dynamic shortest path algorithms.

11.4 A Software Library for Dynamic Graph Algorithms

A systematic effort to build a software repository of implementations of dynamic graph algorithms has been recently initiated in [11.5].

A library of dynamic algorithms has been developed, written in `C++` using LEDA, and is provided as the *LEDA Extension Package on Dynamic Graph Algorithms* (LEP-DGA). The library includes several implementations of simple and sophisticated dynamic algorithms for connectivity, minimum spanning trees, single-source and all-pairs shortest paths, and transitive closure. Actually, the afore mentioned implementations of dynamic connectivity in [11.3] (cf. Section 11.2.1.2.a), dynamic minimum spanning tree in [11.4] (cf. Section 11.2.2.2.a), dynamic transitive closure in [11.32, 11.33] (cf. Section 11.3.1.2.a), and dynamic single-source shortest paths in [11.28] (cf. Section 11.3.2.2.a), are part of the LEP-DGA.

All implementations in the library are accompanied by several demo programs, experimentation platforms, as well as correctness checkers. The library is easily adaptable and extensible, and is available for non-commercial use from `http://www.mpi-sb.mpg.de/LEDA/friends/dyngraph.html`.

All dynamic data structures in the LEP-DGA are implemented as `C++` classes derived from a common base class `dga_base`. This base class defines a common interface for all dynamic algorithms. Except for the usual goals of efficiency, ease of use, extensibility, etc, special attention has been drawn on some domain specific design issues. Two main problems arose in the implementation of the library.

- *Missing Update Operations:* Dynamic algorithms usually support only a subset of all possible update operations, e.g., most dynamic graph algorithms cannot handle single vertex deletions and insertions.
- *Maintaining Consistency:* In an application, a dynamic graph algorithm D may run in the background while the graph changes due to a procedure P which is not aware of D. Consequently, there has to be a means of keeping D consistent with the current graph, because P will not use a possible interface for changing the graph provided by D, but will use the graph directly. Whether D exists or not should have no impact on P.

It was decided to support all update operations for convenience. Those updates which are not supported by the theoretical background are implemented by reinitializing the data structure for the new graph. This may not be very efficient, but it is better than exiting the whole application. The documentation tells the users which updates are supported efficiently or not. The fact that the user calls an update which theoretically is not supported results only in a (perhaps very small) performance penalty. This enhances the robustness of the applications using the library or alternatively reduces the complexity of handling exceptional situations.

An obvious approach to maintain consistency between a graph and a dynamic data structure D working on that graph is to derive D from the graph class. However, this may not be very flexible. In the case where there are more than one dynamic graph data structures working on the same graph, things could get quite complicated with this approach. Instead, the following

approach was used, motivated by the observer design pattern of Gamma et al. [11.34]. A new graph type `msg_graph` has been created which sends messages to interested third parties whenever an update occurs. The base class `dga_base` of all dynamic graph algorithms is one such third party; it receives these messages and calls the appropriate update operations which are virtual methods appropriately redefined by the specific implementations of dynamic graph algorithms.

11.5 Conclusions

We have surveyed several experimental studies which investigate the practicality of dynamic algorithms for fundamental problems in graphs. These studies try to exhibit advantages and limitations of important techniques and algorithms, and to identify the best algorithm for a given input.

In all studies considered, it was evident that sophisticated engineering and fine-tuning of dynamic algorithms is often required to make them competitive or better than simpler, pseudo-dynamic approaches based on static algorithms. Moreover, there were cases where the simpler approaches cannot be beaten by any dynamic algorithm.

In an attempt to draw some rough conclusions on the practicality of dynamic algorithms, we could say that for problems in non-sparse unstructured (random) inputs involving either undirected or directed graphs and operation sequences that are not very small, the dynamic algorithms are usually better than simpler, pseudo-dynamic approaches. In the case of more structured (non-random) inputs, there is a distinction in the behaviour depending on whether the input graph is directed or not. In the latter case, the dynamic algorithms dominate the simpler approaches, while in the former we witness a reverse situation (the simpler algorithms outperform the dynamic ones).

The experimental methodology followed in most papers allows us to sketch some rough guidelines that could be useful in future studies:

- The data sets should be carefully designed to include both unstructured (random) inputs and more structured inputs that include semi-random inputs, pragmatic inputs, and worst-case inputs.
- In any given data set, several values of the input parameters (e.g., number of vertices and edges, length of the operation sequence) should be considered. It was clear from the surveyed experimental studies that several algorithms do not exhibit a stable behaviour and their performance depends on the input parameters. For example, most update bounds are amortized; consequently, the length of the operation sequence turns out to be an important parameter as it clearly determines how well the update bound is amortized in the conducted experiment. In all cases, the measured quantities (usually the CPU time) should be averaged over several samples in order to reduce variance.

- It is important to carefully select the hardware platform upon which the experiments will be carried out. This does not only involve memory issues that eventually appear when dealing with large inputs, but also allows investigation of the practical performance of dynamic algorithms on small inputs. For example, in the latter case it is often necessary to resort to slower machines in order to be able to exhibit the difference among the algorithms.

The experimental methodology followed and the way the test suites developed and evolved in the various studies (usually building upon and enhancing previous test sets) constitute an important guide for future implementors and experimenters of dynamic graph algorithms.

Acknowledgments

The author would like to thank the anonymous referees for several helpful suggestions and comments that improved the paper. The author is also indebted to Umberto Ferraro and Pino Italiano for various clarifications regarding their work.

References

11.1 S. Abdeddaim. On incremental computation of transitive closure and greedy alignment. In *Proceedings of the 8th Symposium on Combinatorial Pattern Matching (CPM'97)*. Springer Lecture Notes in Computer Science 1264, pages 167-179, 1997.

11.2 S. Abdeddaim. Algorithms and experiments on transitive closure, path cover and multiple sequence alignment. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX'00)*, pages 157–169, 2000.

11.3 D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal of Experimental Algorithmics*, 2:5, 1997. Preliminary version in *Proceedings of SODA'96*.

11.4 G. Amato, G. Cattaneo, and G. F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 314–323, 1997.

11.5 D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A software library of dynamic graph algorithms. In *Proceedings of the Workshop on Algorithms and Experiments (ALEX'98)*, pages 129–136, 1998.

11.6 C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th Symposium on Foundations of Computer Science (FOCS'89)*, pages 540–545, 1989.

11.7 G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12:615–638, 1991.

11.8 A. Bateman, E. Birney, R. Durbin, S. Eddy, K. Howe, and E. Sonnhammer. The PFAM protein families database. *Nucleid Acids Research*, 28:263–266, 2000.

11.9 B. Bollobás. *Random Graphs*. Academic Press, New York, 1985.

11.10 G. Cattaneo, P. Faruolo, U. Ferraro-Petrillo, and G. F. Italiano. Maintaining dynamic minimum spanning trees: an experimental study. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*. Springer Lecture Notes in Computer Science, to appear.

11.11 S. Chaudhuri and C. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: sequential algorithms. *Algorithmica*, 27:212–226, 2000.

11.12 S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi dynamic problems in digraphs. *Theoretical Computer Science*, 203 (1):69–90, 1998.

11.13 C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS'00)*, pages 381–389, 2000.

11.14 C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 260–267, 2001.

11.15 C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U.Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: an experimental study. In *Proceedings of the 4th Workshop on Algorithm Engineering (WAE'00)*. Springer Lecture Notes in Computer Science 1982, pages 218–229, 2000.

11.16 E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

11.17 H. Djidjev, G. Pantziou, and C. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28:367–389, 2000.

11.18 J. Edmonds and R. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

11.19 D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44:669–696, 1997. Preliminary version in *Proceedings of FOCS'92*.

11.20 S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

11.21 S. Even and Y. Shiloach. An on-line edge deletion problem. *Journal of the ACM*, 28:1–4, 1981.

11.22 P. Fatourou, P. Spirakis, P. Zarafidis, and A. Zoura. Implementation and experimental evaluation of graph connectivity algorithms using LEDA. In *Proceedings of the 3rd Workshop on Algorithm Engineering (WAE'99)*. Springer Lecture Notes in Computer Science 1668, pages 124–138, 1999.

11.23 U. Ferraro-Petrillo. Personal Communication, February 2002.

11.24 P. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic shortest paths and breadth-first search on digraphs. In *Proceedings of the Symposium on Theorecial Aspects of Computer Science (STACS'97)*. Springer Lecture Notes in Computer Science 1200, pages 33–46, 1997.

11.25 G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14:781–798, 1985.

11.26 G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computing (FOCS'91)*, pages 632–641, 1991.

11.27 M. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.
11.28 D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *ACM Journal of Experimental Algorithmics*, 3:5, 1998.
11.29 D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22(3):250–274, 1998.
11.30 D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *Proceedings of the 6th European Symposium on Algorithms (ESA'98)*. Springer Lecture Notes in Computer Science 1461, pages 320-331, 1998.
11.31 D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):351–381, 2000. Preliminary version in *Proceedings of SODA'96*.
11.32 D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schäfer, and C. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *Proceedings of the 6th European Symposium on Algorithms (ESA'98)*. Springer Lecture Notes in Computer Science 1461, pages 368-380, 1998.
11.33 D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithmics*, 6, 2001.
11.34 E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison-Wesley, 1995.
11.35 D. Harel. On-Line maintenance of the connected components of dynamic graphs. Manuscript, 1982.
11.36 M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 27th ACM Symposium on Theory of Computing (STOC'95)*, pages 519–527, 1995.
11.37 M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 664–672, 1995.
11.38 M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming (ICALP'97)*. Springer Lecture Notes in Computer Science 1256, pages 594-604, 1997.
11.39 M. R. Henzinger and M. Thorup. Sampling to provide or to bound: with applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11:369–379, 1997. Preliminary version in *Proceedings of ICALP'96*.
11.40 J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the 30th ACM Symposium on Theory of Computing (STOC'98)*, pages 79–89, 1998.
11.41 T. Ibaraki and N. Katoh. On-line computation of transitive closure of graphs. *Information Processing Letters*, 16:95–97, 1983.
11.42 G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.
11.43 G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28:5–11, 1988.
11.44 R. Iyer, D. Karger, H. Rahul, and M. Thorup. An experimental study of polylogarithmic fully-dynamic connectivity algorithms. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX'00)*, pages 59–78, 2000.

11.45 H. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, 15(4):558–598, 1990.
11.46 S. Khanna, R. Motwani, and R. Wilson. On certificates and lookahead in dynamic graph problems. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pages 222–231, 1996.
11.47 V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 81–91, 1999.
11.48 V. King, and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the 31st ACM Symposium on Theory of Computing (STOC'99)*, pages 492–498, 1999.
11.49 J. A. La Poutré, and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proceedings of the 14th Workshop on Graph-Theoretic Concepts in Computer Science (WG'88)*. Springer Lecture Notes in Computer Science 314, pages 106–120, 1988.
11.50 P. Loubal. A network evaluation procedure. *Highway Research Record*, 205 (1967):96-109.
11.51 K. Mehlhorn. *Data Structures and Algorithms. Vol. 2: Graph Algorithms and NP-Completeness.* Springer-Verlag, 1984.
11.52 K. Mehlhorn and S. Näher. *LEDA — A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.
11.53 P. B. Miltersen, S. Subramanian, J. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.
11.54 J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical Report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.
11.55 S. Nikoletseas, J. Reif, P. Spirakis, and M. Yung. Stochastic graphs have short memory: fully dynamic connectivity in poly-log expected time. In *Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming (ICALP'95)*. Springer Lecture Notes in Computer Science 944, pages 159–170, 1995.
11.56 G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
11.57 G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-paths problem. *Journal of Algorithms*, 21:267–305, 1996.
11.58 The parametric problem of shortest distances. *USSR Computational Mathematics and Mathematical Physics*, 8(5):336-343, 1968.
11.59 H. Rohnert. A dynamization of the all pairs least cost path problem. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS'85)*. Springer Lecture Notes in Computer Science 182, pages 279–286, 1985.
11.60 K. Simon. An improved algorithm for transitive closure on acyclic graphs. *Theoretical Computer Science*, 58:325–346, 1988.
11.61 D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 24:362–381, 1983.
11.62 R. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22:215–225, 1975.
11.63 J. Thompson, F. Plewniak, and O. Poch. BAliBASE: a benchmark alignments database for evaluation of multiple sequence alignment programs. *Bioinformatics*, 15:87–88, 1999.

11.64 M. Thorup. Decremental dynamic connectivity. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 305–313, 1997.
11.65 M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*, pages 343–350, 2000.
11.66 D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

Author Index

Bader, David A. 1

Cohen, Paul R. 93

Demetrescu, Camil 24

Fellows, Michael R. 51
Finocchi, Irene 24
Fleischer, Rudolf 93
Fortna, Ray 78

Italiano, Giuseppe F. 24

Ladner, Richard E. 78

McGeoch, Catherine 93
Meinel, Christoph 127, 139
Moret, Bernard M. E. 1, 163

Näher, Stefan 24
Nguyen, Bao-Hoang 78

Precup, Doina 93

Sack, Harald 127
Sanders, Peter 1, 93, 181
Spirakis, Paul 197
Stangier, Christian 139

Wagner, Arno 127
Warnow, Tandy 162

Zaroliagis, Christos D. 197, 229